Distributed Control Flow with Classical Modal Logic

Conference Paper *in* Lecture Notes in Computer Science · July 2005

DOI: 10.1007/11538363_6 · Source: CiteSeer

Tom Murphy VII
Carnegie Mellon University
22 PUBLICATIONS **304** CITATIONS
SEE PROFILE

Robert Harper
PortaScience
167 PUBLICATIONS **6,015** CITATIONS
SEE PROFILE

Distributed Control Flow with Classical Modal Logic

Tom Murphy VII **Karl Crary**
Robert Harper

December 14, 2004
CMU-CS-04-177

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

In previous work we presented a foundational calculus for spatially distributed computing based on intuitionistic modal logic. Through the modalities $\Box$ and $\Diamond$ we were able to capture two key invariants: the mobility of portable code and the locality of fixed resources.

This work investigates issues in distributed control flow through a similar propositions-as-types interpretation of *classical* modal logic. The resulting programming language is enhanced with the notion of a network-wide continuation, through which we can give computational interpretation of classical theorems (such as $\Box A \equiv \neg\Diamond\neg A$). Such continuations are also useful primitives for building higher-level constructs of distributed computing. The resulting system is elegant, logically faithful, and computationally reasonable.

The ConCert Project is supported by the National Science Foundation under grant ITR/SY+SI 0121633: "Language Technology for Trustless Software Dissemination".

Keywords: distributed computing, modal logic, classical logic, continuations, programming languages

1 Introduction

This paper is an exploration of distributed control flow using a propositions-as-types interpretation of classical modal logic. We build on our previous intuitionistic calculus, Lambda 5 [8], which is a simple programming language (and associated logic) for distributed computing. Lambda 5 focuses particularly on the spatial distribution of programs, and allows the programmer to express the *place* in which computation occurs using modal typing judgments. Through the modal operators $\Box$ and $\Diamond$ we are then able to express invariants about mobility and locality of resources. Our new calculus, C5, extends Lambda 5 with network-wide first-class continuations, which arise naturally from the underlying classical logic. Network-wide continuations create a new relationship between the modalities $\Box$ and $\Diamond$, which we see with several examples, and serve as building-blocks for other useful primitives. Before we introduce C5, we begin with a short reprise of Lambda 5.

1.1 Lambda 5

The Lambda 5 programming model is a network with many different *places*, or *nodes*. In order to be faithful to this model, we use a style of logic that has the ability to reason simultaneously from multiple perspectives, namely, modal logic. Compared to propositional logic, which is concerned with *truth*, modal logic deals with truth at different *worlds*. These worlds are related by an *accessibility relation*, which affects the strength of the modal connectives; different assumptions about accessibility give rise to different modal logics. For modelling a network where the worlds are nodes, we choose Intuitionistic S5 [15], whose relation is reflexive, symmetric, and transitive—every world is related to every other world. Therefore, except when comparing it to other systems, we essentially dispense with the accessibility relation altogether. This leads to a simple explanation of the judgments and connectives, which is as follows.

$A\,\mathsf{true}\ @\omega$ is the basic judgment, meaning that the proposition A is true at the world ω (we abbreviate this to $A@\omega$). There are two new proposition forms for quantifying over worlds. $\Box A$ is the statement that A is true at every world. $\Diamond A$ means that A is true at *some* world. Because we think of these worlds as places in the network, operationally we interpret type $\Box A$ as representing mobile code or data of type A, and the type $\Diamond A$ as an address of code or data of type A.

Propositions must be situated at a world in order to be judged true, so it is important to distinguish between the proposition $\Box A$ and the judgment $\Box A@\omega$, the latter meaning that A is true in every world *from the perspective of* ω. In S5, every world has the same perspective with regard to statements about *all* or *some* world(s). But operationally this will be significant, as there is no true "global" code, only mobile code that currently exists at some world.

Though the logic distinguishes between $\Box A@\omega$ and $\Box A@\omega'$, both have precisely the same immediate consequences. The typical rule for eliminating $\Box$, for instance as given by Simpson [15] is

$$\frac{\Box A@\omega}{A@\omega'}$$

With this rule, it *never really matters* where $\Box A$ exists, since we can eliminate it instantly to any world. However, we really do care operationally where mobile code resides, and so we adjust the rules of Lambda 5 to reflect this bias. The logic features a novel *decomposition* into locally-acting introduction and elimination rules, i.e.

$$\frac{\Box A \,@\, w}{A \,@\, \omega}$$

as well as *motion* rules for moving between worlds, i.e.

$$\frac{\Box A \,@\, \omega}{\Box A \,@\, \omega'}$$

We argue [8] that this results in a more appropriate operational interpretation. Our classical system also features this decomposition, and like Lambda 5, we are able to retain a crisp connection to the underlying logic.

Although distributed computing problems are often thought of as being concurrent, both Lambda 5 and our new calculus are sequential. We consider concurrency an orthogonal issue, although we give remarks on how it can be accomplished in Section 6.

1.2 Classical Control Flow

The notion that control operators such as Scheme's `call/cc` or Felleisen's $\mathcal{C}$ can be given logical meaning via classical logic is well known. Essentially, if we interpret the type $\neg A$ as a continuation expecting a value of type A, then the types of these operators are classical tautologies. Griffin first proposed this in 1990 [5] with later refinements by (for example) Murthy [9]. Parigot's $\lambda\mu$-calculus [10] takes this idea and develops it into a full-fledged natural deduction system for classical logic.[1] It began to become clear that this was no accident—classical logic *is* the logic of control flow.

Therefore, a natural next step is to look at *classical* S5 to see what kind of programming language it gives us, which is the topic of this paper. We find that the notion of a network-wide continuation arises naturally, giving a computational explanation to (intuitionistically ridiculous) classical theorems such as $\Box A \equiv \neg\Diamond\neg A$. We also believe that such primitives can be useful for building distributed computing mechanisms such as synchronous message passing.

The paper proceeds as follows. We first present classical S5 judgmentally, giving a natural deduction system and intuition for its operational behavior. We then verify that our proof system really is faithful by establishing a correspondence with a dual sequent calculus that admits cut. Next we give proof terms for some classical theorems, to elucidate the new connection between $\Box$ and $\Diamond$ made possible by network-wide continuations. In order to make these intuitions concrete, we then give an operational semantics based on an abstract network. We follow with some ideas about concurrency and how network-wide continuations can be used by distributed applications, and conclude with a discussion of related work.

[1]Our calculus is quite similar to his (extended to the modal case!), although we prefer to present it with an emphasis on *truth* and *falsehood* judgments.

$$\frac{\Gamma, x{:}A \mathbin{@} \omega; \Delta \vdash M : B \mathbin{@} \omega}{\Gamma; \Delta \vdash \lambda x.M : A \supset B \mathbin{@} \omega} \supset I \qquad \frac{\begin{array}{c}\Gamma; \Delta \vdash N : A \mathbin{@} \omega \\ \Gamma; \Delta \vdash M : A \supset B \mathbin{@} \omega\end{array}}{\Gamma; \Delta \vdash MN : B \mathbin{@} \omega} \supset E$$

$$\frac{\Gamma; \Delta \vdash \overset{\overset{\omega'}{\vdots}}{M} : A \mathbin{@} \omega'}{\Gamma; \Delta \vdash \texttt{box}\ \omega'.M : \Box A \mathbin{@} \omega} \Box I \qquad \frac{\Gamma; \Delta \vdash M : \Box A \mathbin{@} \omega}{\Gamma; \Delta \vdash \texttt{unbox}\, M : A \mathbin{@} \omega} \Box E$$

$$\frac{}{\Gamma, x{:}A \mathbin{@} \omega, \Gamma'; \Delta \vdash x : A \mathbin{@} \omega} \text{hyp} \qquad \frac{\Gamma; \Delta \vdash M : \Box A \mathbin{@} \omega' \quad \omega'}{\Gamma; \Delta \vdash \texttt{get}_{\Box}[\omega']M : \Box A \mathbin{@} \omega} \Box M$$

$$\frac{\begin{array}{c}\Gamma, x{:}A \mathbin{@} \omega'; \overset{\overset{\omega'}{\vdots}}{\Delta} \vdash N : B \mathbin{@} \omega \\ \Gamma; \Delta \vdash M : \Diamond A \mathbin{@} \omega\end{array}}{\Gamma; \Delta \vdash \texttt{letd}\, \omega'.x = M\, \texttt{in}\, N : B \mathbin{@} \omega} \Diamond E \qquad \frac{\Gamma; \Delta \vdash M : A \mathbin{@} \omega}{\Gamma; \Delta \vdash \texttt{here}\, M : \Diamond A \mathbin{@} \omega} \Diamond I$$

$$\frac{\Gamma; \Delta \vdash M : \Diamond A \mathbin{@} \omega' \quad \omega'}{\Gamma; \Delta \vdash \texttt{get}_{\Diamond}[\omega']M : \Diamond A \mathbin{@} \omega} \Diamond M \qquad \frac{\Gamma; \Delta, u{:}A \star \omega \vdash M : A \mathbin{@} \omega}{\Gamma; \Delta \vdash \texttt{letcc}\, u\, \texttt{in}\, M : A \mathbin{@} \omega} \text{bc}$$

$$\frac{\Gamma; \Delta, u{:}A \star \omega \vdash M : A \mathbin{@} \omega}{\Gamma; \Delta, u{:}A \star \omega \vdash \texttt{throw}\, M\, \texttt{to}\, u : C \mathbin{@} \omega'} \# \qquad \frac{\Gamma; \Delta \vdash M : \bot \mathbin{@} \omega' \quad \omega'}{\Gamma; \Delta \vdash \texttt{rpc}[\omega']M : C \mathbin{@} \omega} \bot E$$

$$\frac{\Gamma; \Delta \vdash M : A \mathbin{@} \omega \quad \Gamma; \Delta \vdash N : B \mathbin{@} \omega}{\Gamma; \Delta \vdash \langle M, N \rangle : A \wedge B \mathbin{@} \omega} \wedge I \qquad \frac{\Gamma; \Delta \vdash M : A_1 \wedge A_2 \mathbin{@} \omega}{\Gamma; \Delta \vdash \pi_i M : A_i \mathbin{@} \omega} \wedge E_i$$

Figure 1: Classical S5 natural deduction

All of the proofs in this paper have been formalized in the Twelf system [12] and verified by its metatheorem checker [14].[2]

2 Classical S5

Because we wish to take a propositions-as-types interpretation of modal logic, a judgmental proof theory for our logic is critical. In this section we give such a presentation of Classical S5.

Because modal logic is concerned with truth relativized to worlds, our judgments must reflect that. We have two main judgments in our proof theory. The first,

$$A\,\mathsf{true} \mathbin{@} \omega$$

simply states that the proposition A is true at the world ω. Dually, we have

$$A\,\mathsf{false} \star \omega$$

[2]They can be found at `http://www.cs.cmu.edu/~concert/`.

which says that the proposition A is *false* at the world ω. Although these two judgments are dual, the natural deduction system is biased towards the first; it is primarily concerned with deducing that propositions are true. We will only make assumptions about falsehood for the purpose of deriving a contradiction. As is standard, we reify the hypotheses about truth and falsehood into contexts (eliding true and false), and the central judgment of our proof theory becomes

$$\Gamma; \Delta \vdash A @ \omega$$

where we deduce that A is true at world ω under truth assumptions of the form $B @ \omega'$ appearing in Γ and falsehood assumptions of the form $C \star \omega''$ appearing in Δ. We also have hypotheses about the existence of worlds. It is cumbersome to write a context of world assumptions and conditions on world existence in every judgment. Instead we use pure hypothetical notation

$$\begin{array}{c} \omega' \\ \vdots \\ \Gamma; \Delta \vdash A @ \omega \end{array}$$

to express a judgment hypothetical in the existence of world ω' (which may be the same as ω!). We also take the common shortcut of only permitting mention of worlds that exist. Therefore, all judgments are hypothetical in at least *some* world (the world at which the conclusion is formed), until we introduce world constants in Section 5.

Operationally, we will think of a falsehood assumption $A \star \omega$ as a continuation, living at world ω, that expects something of type A.

Our natural deduction system appears in Figure 1. These rules include proof terms, which we will explain shortly. Aside from the falsehood context, the rules for $\Box$, $\Diamond$ and $\supset$ are the same as in Lambda 5. The new connectives $\bot$ (discussed below) and $\wedge$ are treated as they would be in the intuitionistic case. The major additions are the structural rules *bc* (by contradiction) and # (contradict), which enable classical reasoning.

The *bc* rule is read as follows: In order to prove $A @ \omega$, we can assume that A is false at ω. This corresponds directly to the classical axiom $(\neg A \supset A) \supset A$. Operationally, this grabs the current continuation and binds the falsehood variable to it. The # rule may be alarming at first glance, because it requires the assumption $A \star \omega$ to appear in the conclusion. This is because the # rule is actually the *hypothesis* rule for falsehood assumptions, and will have a corresponding substitution principle.[3] The rule simply states that if we have the assumption that A is false and are able to prove that A is true (at the same world), then we can deduce a contradiction and thus any proposition. The # rule is realized operationally as a `throw` of an expression to a matching continuation. Note that continuations are *global*—we can throw from any world to a remote continuation $A \star \omega$, provided that we are able to construct a proof of $A @ \omega$.

The rules for $\Box$ and $\Diamond$ are important to review. $\Box$ elimination is the easiest to understand: If we know that $\Box A$ is true at some world, then we know A is true at the same world. To prove $\Box A$, we must prove A at a hypothetical world about which nothing is known (rule $\Box I$). Operationally, we realize $\Box A$ as a piece of suspended code, with the hypothetical world ω' bound within it. Introduction of $\Diamond$ is simple; if we know A then we know that A is true

[3] A theory of *hypothetical hypotheticals* would be able to express this in a less awkward—but perhaps no less alarming—way. Abel [1] for instance gives such a third-order encoding of the $\lambda\mu$-calculus.

somewhere (namely here). Operationally this will record the value in a table and return an address that witnesses its existence. Elimination of $\Diamond$ is as follows: if we know $\Diamond A$, then we know there is some world where A is true (but we don't know anything else about it). Call this world ω' and assume $A\,@\omega'$ in order to continue reasoning. Finally, we provide *motion* rules (as per our decomposition) $\Box M$ and $\Diamond M$. Both simply allow knowledge of $\Box A$ or $\Diamond A$ at one world to be transported to another. Operationally these move the values between worlds.

Bottom has no introduction form, but we allow the *remote elimination* of it (rule $\bot E$). This is similar to the motion rules for $\Box$ and $\Diamond$, except that nothing is ever returned, because there is no canonical value of type $\bot$. For this reason we call the proof term `rpc`, as it invokes a sort of remote procedure call on the target world.[4]

For each kind of hypothesis we have a substitution theorem.

Theorem 1 (Truth Substitution)
If $\Gamma; \Delta \vdash M : A\,@\omega$
and $\Gamma, x{:}A\,@\omega; \Delta \vdash N : B\,@\omega'$
then $\Gamma; \Delta \vdash [M/x]N : B\,@\omega'$.

Theorem 2 (Falsehood Substitution)
If $\forall C, \omega''.\ \Gamma, x{:}A\,@\omega; \Delta \vdash M : C\,@\omega''$
and $\Gamma; \Delta, u{:}A\star\omega \vdash N : B\,@\omega'$
then $\Gamma; \Delta \vdash [\![x.M/u]\!]N : B\,@\omega'$.

Here, truth substitution $[M/x]N$ is defined in the standard way. Note again that a proof of type $B\,@\omega'$ can contain sub-expressions that are well-typed at *other* worlds! In the operational semantics we will always ship these expressions to their home world before evaluating them. Theorem 2, however, warrants special attention. This principle is dual to the # rule just as Theorem 1 is dual to *hyp*. The # rule contradicts an $A\star\omega$ with an $A\,@\omega$, so to eliminate a falsehood assumption by substitution we are able to assume $A\,@\omega$ and must produce another contradiction. Reading $\vdash$ as logical consequence, we have that if $A\,\mathsf{false}$ gives B, and $A\,\mathsf{true}$ gives C (for all C), then B. This can easily be seen as a consequence of excluded middle. We write this substitution as $[\![x.M/u]\!]N$ where x is a binder (with scope through M) that stands for the value thrown to u. It is defined pointwise on N except for a use of the # rule on u:

$$[\![x.M/u]\!]\,\texttt{throw}\,N'\,\texttt{to}\,u \;\doteq\; [N'/x]M$$

This principle is close to what Parigot calls *structural substitution* for the $\lambda\mu$-calculus. Operationally, we see this as replacing the throw with some other handler for A. Since the new handler must have parametric type, typically it is a `throw` to some other continuation, perhaps after performing some computation on the proof of A.

Proof of Theorem 2 is by a straightforward induction on the derivation of B, appealing to Theorem 1 in the case above. □

[4]We could have equivalently had a $\texttt{get}_\bot$ and a local `abort`, but there appears to be no practical use to this decomposition.

$$\dfrac{}{\Gamma, A\,@\,\omega \;\#\; A\star\omega, \Delta}\;\text{contra} \qquad \dfrac{}{\Gamma, \bot\,@\,\omega \;\#\; \Delta}\;\bot T$$

$$\dfrac{\begin{array}{c}\Gamma, A \supset B\,@\,\omega, B\,@\,\omega \;\#\; \Delta \\ \Gamma, A \supset B\,@\,\omega \;\#\; A\star\omega, \Delta\end{array}}{\Gamma, A \supset B\,@\,\omega \;\#\; \Delta}\;\supset T \qquad \dfrac{\Gamma, A\,@\,\omega \;\#\; B\star\omega, A \supset B\star\omega, D}{\Gamma \;\#\; A \supset B\star\omega, D}\;\supset F$$

$$\dfrac{\Gamma, \Box A\,@\,\omega, A\,@\,\omega' \;\#\; \Delta}{\Gamma, \Box A\,@\,\omega \;\#\; \Delta}\;\Box T \qquad \dfrac{\begin{array}{c}\omega' \\ \vdots \\ \Gamma \;\#\; A\star\omega', \Box A\star\omega, \Delta\end{array}}{\Gamma \;\#\; \Box A\star\omega, \Delta}\;\Box F$$

$$\dfrac{\begin{array}{c}\omega' \\ \vdots \\ \Gamma, \Diamond A\,@\,\omega, A\,@\,\omega' \;\#\; \Delta\end{array}}{\Gamma, \Diamond A\,@\,\omega \;\#\; \Delta}\;\Diamond T \qquad \dfrac{\Gamma \;\#\; A\star\omega', \Diamond A\star\omega, \Delta}{\Gamma \;\#\; \Diamond A\star\omega, \Delta}\;\Diamond F$$

$$\dfrac{\Gamma, A \wedge B\,@\,\omega, A\,@\,\omega, B\,@\,\omega \;\#\; \Delta}{\Gamma, A \wedge B\,@\,\omega \;\#\; \Delta}\;\wedge T \qquad \dfrac{\begin{array}{c}\Gamma \;\#\; A\star\omega, A \wedge B\star\omega, \Delta \\ \Gamma \;\#\; B\star\omega, A \wedge B\star\omega, \Delta\end{array}}{\Gamma \;\#\; A \wedge B\star\omega, \Delta}\;\wedge F$$

Figure 2: Classical S5 sequent calculus

We wish to know that our proof theory (specially constructed to give rise to a good operational semantics) is not simply ad hoc; that it really embodies classical S5, and is globally sound. To do so we prove in the next section a correspondence to a straightforward sequent formulation of classical S5 with the subformula property. We'll use the sequent calculus as intuition as we develop proof terms for some classically true propositions. Following that is a discussion of the operational semantics (Section 5), which does not depend on the sequent calculus.

3 Sequent Calculus

Our sequent calculus is motivated by simplicity and duality alone, because we will not give it a computational interpretation. One traditional way of doing classical theorem proving is to negate the target formula and prove a contradiction from it. We base our sequent calculus around this view: the sequent

$$\Gamma \;\#\; \Delta$$

means that the truth assumptions in Γ and the falsehood assumptions in Δ are mutually contradictory.[5] The calculus is given in Figure 2. We treat contexts as unordered multisets, so the *action* can occur anywhere in either context.

These rules should be read bottom-up, as if during proof search. The *contra* rule allows us to form a contradiction whenever a proposition is both true and false at the same world. The $\Box T$ rule says that if we know $\Box A\,@\,\omega$, then we know $A\,@\,\omega'$ for any ω' that exists. On the

[5]Our rules are also consistent with the more typical multiple-conclusion reading, "if all of Γ are true, then one of Δ is true."

other hand, if we know that $\Box A$ is false, then we know A is false at some world ω'. However, we must treat this world as hypothetical and fresh since we don't know which one it is. The rules for $\Diamond$ are perfect mirror images of the rules for $\Box$. For implication, we use the classical truth tables to provide rules of inference. If we know that $A \supset B$ is false, then we know A is true but B is false. If we know that $A \supset B$ is true, then we know that *either* A is false or B is true.

A key feature of the sequent calculus is the subformula property: every step (when read bottom-up) proceeds by decomposing exactly one connective. This means that proofs in the sequent calculus work by only examining the structure of the proposition at hand; this gives us a nice orthogonality condition for the connectives in our logic.

The translation from natural deduction to the sequent calculus requires a lemma (which should not be a rule of inference because it violates the subformula property). In an intuitionistic calculus this would be *cut*; for the symmetric classical calculus it turns out to be the familiar classical notion of *excluded middle.*

Theorem 3 (Excluded Middle)

If $\Gamma, A\,@\omega \;\#\; \Delta$
and $\Gamma \;\#\; A{\star}\omega, \Delta$
then $\Gamma \;\#\; \Delta.$

Proof of Theorem 3 is by lexicographic induction on the proposition A and then on the two derivations. □

With excluded middle, we can prove the correspondence between natural deduction and the sequent calculus.

Theorem 4 (Equivalence)

(a) *If* $\Gamma; \Delta \vdash M : A\,@\omega$
then $\Gamma \;\#\; A{\star}\omega, \Delta.$

(b) *If* $\Gamma \;\#\; \Delta$
then $\exists M.\ \forall C, \omega.\ \Gamma; \Delta \vdash M : C\,@\omega.$

It is easy to see why 4(b) is the right statement. Since we think of $\Gamma \;\#\; \Delta$ as a proof of contradiction, this corresponds to a derivation that proves any proposition at any world in natural deduction. Theorem 4(a) is more subtle. We show that if A is true under assumptions Γ and Δ, then A being false at the same world is contradictory with those assumptions. Computationally, we can think of this as the "final continuation" to which the result computed in natural deduction is passed. Putting these two theorems together, we have that $\Gamma; \Delta \vdash M : A\,@\omega$ gives $\Gamma \;\#\; A{\star}\omega, \Delta$, which then gives $\forall C, \omega'.\ \Gamma; \Delta, u{:}A{\star}\omega \vdash M' : C\,@\omega'$. In particular, we choose $C = A$ and $\omega' = \omega$, and then by application of *bc* we have the original judgment (with perhaps a different proof term $\texttt{letcc}\, u \,\texttt{in}\, M'$). Thus $\vdash$ and $\#$ are really equivalent.

The proof of Theorem 4(a) is by straightforward induction on the derivation, using Theorem 3 where necessary. (The structural rules *bc* and $\#$ just become uses of contraction and weakening in the sequent calculus.) Proof of 4(b) is tricker. Uses of T rules are easy; they correspond directly to the elimination rules[6] in natural deduction. But since our natural deduction is biased towards manipulating truth rather than falsehood, the F rules are more difficult and make nontrivial use of the falsehood substitution theorem. For instance, in the $\wedge F$ case we have by induction:

$$\begin{array}{c}\Gamma; \Delta, up{:}A \wedge B \star \omega, ua{:}A \star \omega \vdash N_1 : C \,@\, \omega' \quad (\forall C, \omega') \\ \Gamma; \Delta, up{:}A \wedge B \star \omega, ub{:}B \star \omega \vdash N_2 : C \,@\, \omega' \quad (\forall C, \omega')\end{array}$$

By two applications of Theorem 2, we get that the following proof term has any type at any world:

$$\Big[\!\Big[x. [\![y.\, \texttt{throw}\, \langle x, y \rangle\, \texttt{to}\, up / ub]\!] N_2 \,/ ua \Big]\!\Big] N_1$$

First, we form a `throw` of the pair $\langle x, y \rangle$ to our pair continuation up. This has free truth hypotheses $x : A$ and $y : B$. Therefore, we can use it to substitute away the ub continuation in N_2 (any throw of M to ub becomes a throw of $\langle x, M \rangle$ to up). Finally, we can use this new term to substitute away ua in N_1, giving us a term that depends only on the pair continuation up. This pattern of *prepending* work onto continuations through substitution is characteristic of this proof, and reflects our bias towards the truth judgment in natural deduction. As another example, in the case for the $\Diamond F$ rule we have by induction:

$$\Gamma; \Delta, u{:}A \star \omega', ud{:}\Diamond A \star \omega \vdash N : C \,@\, \omega'' \quad (\forall C, \omega'')$$

Our proof term in natural deduction is then:

$$[\![x.\, \texttt{throw}(\texttt{get}_{\Diamond}[\omega'](\texttt{here}\, x))\, \texttt{to}\, ud / u]\!] N$$

Simply enough, if u is ever thrown to, then we instead take that term's address (which lives at ω'), move it to ω, and throw it to our $\Diamond A$ continuation ud.

Finally, the case for $\Box F$ is interesting because it involves a `letcc`.[7] By induction we have:

$$\forall \omega'.\;\; \Gamma; \Delta, u{:}A \star \omega', ub{:}\Box A \star \omega \vdash N : C \,@\, \omega'' \quad (\forall C, \omega'')$$

Then the proof term witnessing the theorem here is:

$$\texttt{throw}(\texttt{box}\; \omega'.\, \texttt{letcc}\, u\, \texttt{in}\, N)\, \texttt{to}\, ub$$

It is not possible to use falsehood substitution on u in this case. To do so we would need to turn a term of type $A @ \omega'$ into a $\Box A @ \omega$ to throw to ub. Although at a meta-level we know that we can choose any ω', it won't be possible to internalize this in order to create a $\Box A$. Instead we must introduce a new box, and choose ω' to be the new hypothetical world that the $\Box I$ rule introduces. At that point we use `letcc` to create a real $A \star \omega'$ assumption to discharge u. The remaining cases are similar or straightforward, and can be found in full detail in the Twelf code, which can be found in Appendix A.[8] □

[6]Except for implication, which is phrased differently in the sequent calculus.

[7]In fact, this is the only place in the proof where a `letcc` is necessary. This suggests a normal form for natural deduction terms where `letcc` appears only once at the outermost scope and immediately inside each `box`.

[8]The most natural LF encoding of falsehood is 3rd-order [1]; we use a 2nd-order encoding in our proofs (proving the falsehood substitution theorem by hand) because third-order proof checking is not yet in the distribution.

4 Examples

In this section we give proof terms showing the new connection between $\Box$ and $\Diamond$ made possible by network-wide continuations. Because the examples we'll look at involve negation ($\neg A$), we'll need to briefly explain how we treat it.

4.1 Negation

Although we have not given the rules for the negation connective, it is easily added to the system. Here we equivalently take the standard shortcut of treating $\neg A$ as an abbreviation for $A \supset \bot$. We computationally read $\neg A @ \omega$ as a continuation expecting A, although this should be distinguished from primitive continuations u with type $A \star \omega$: the former is formed by lambda abstraction and eliminated by application, while the latter is formed with `letcc` and eliminated by a `throw` to it. The two are related in that we can reify a continuation assumption $u{:}A \star \omega$ as a negated formula $\neg A$ by lambda abstracting a throw to it: $\lambda a.\, \texttt{throw}\, a\, \texttt{to}\, u$. Likewise, we can get a falsehood assumption from a term M of type $\neg A$, namely $M(\texttt{letcc}\, u\, \texttt{in} \ldots)$.

Finally, note that we have derived sequent calculus rules $\neg T$ and $\neg F$. Each just flips the proposition under negation to the other side of the sequent, as expected. (The reader can verify that these are indeed the rules derived from $\supset T$ and $\supset F$ if the antecedent of implication is $\bot$.)

4.2 Classical Axioms

Our first example comes from the standard practice in classical modal logic of defining $\Box$ in terms of $\Diamond$:

$$\Box A \equiv \neg \Diamond \neg A$$

From left to right the implication is intuitionistically valid, so we'll look at the proof of the implication right to left. We begin with the sequent calculus proof, to show why this is clearly true classically. We elide any residual assumptions that go unused.

$$
\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{}{-, A@\omega' \;\#\; A@\omega', -}\;\text{contra}}{- \;\#\; \neg A@\omega', A@\omega', -}\;\neg F}{- \;\#\; \Diamond\neg A@\omega, A@\omega', -}\;\Diamond F}{- \;\#\; \Diamond\neg A@\omega, \Box A@\omega}\;\Box F^{\omega'}}{\neg\Diamond\neg A@\omega \;\#\; \Box A@\omega}\;\neg T}{\#\; \neg\Diamond\neg A \supset \Box A@\omega}\;\supset F
$$

Critically, we are using $\Box F$ to get the hypothetical world at which $\Box A$ is false. From there, we can learn $\neg A$ at the same world, which leads to a contradiction. In natural deduction, the proof tells an interesting story:

$$
\begin{array}{ll}
\lambda dc. & (dc : \neg\Diamond\neg A @ \omega) \\
\quad \texttt{box}\ \omega'. & (\text{need to return } A) \\
\quad\quad \texttt{letcc}\, u\, \texttt{in}\, \texttt{rpc}[\omega] & (\text{applying } dc \text{ will yield } \bot) \\
\quad\quad\quad dc(\texttt{get}_{\Diamond}[\omega](\texttt{here}(\lambda a.\, \texttt{throw}\, a\, \texttt{to}\, u))) &
\end{array}
$$

In each example, we'll assume that the whole term lives at the world ω. Operationally, the reading of $\neg\Diamond\neg A \supset \Box A$ is that given a continuation *dc* (expecting the address of an A continuation), we will return a boxed A that is well-formed anywhere. The proof term given accomplishes this by creating a box that, when opened, grabs the current continuation u, which has type $A\star\omega'$. With the continuation in hand, we travel back to ω (where *dc* lives), and apply *dc* to the address of a function that throws to u. In short, at the moment the box is opened we have a *lack of* an A, which we can grab with `letcc` and then take the address of with `here`. This is enough to send to the continuation that we're provided.

Dually we can define $\Diamond$ in terms of $\Box$. Again, one direction is intuitionistically valid. The other,

$$\neg\Box\neg A \supset \Diamond A$$

is asked to conjure up an address of an arbitrary A given a continuation. It is implemented by the following proof term:

```
λbc.                         (bc: $\neg\Box\neg A @ \omega$)
 letcc u in                  ($u : \Diamond A \star \omega$)
  bc(box ω'.λa.              ($a : A @ \omega'$)
    throw(get◇[ω'](here a)) to u)
```

Here, we immediately do a `letcc`, grabbing the $\Diamond A$ continuation at ω. We then form a `box` to pass to the continuation *bc*. It contains a function of type $A \supset \bot$, which takes the address of its argument and throws it to the saved continuation u. Thus the location of A that we return is any world that invokes the $\neg A$ that we've boxed up.

We've left disjunction out of our calculus. Theoretically it poses no problem, although operationally it requires us to perform some tricks to avoid a strange "remote case analysis." In Section 7 however, we see that we can encode it using the de Morgan translation into $\neg$ and $\wedge$. Regardless of how we implement it, disjunction is a source of a wealth of interesting programs.

Without being as formal, let's take a look at a program implementing the classical axiom $\Box A \vee \Diamond\neg A$. We'll assume constructors `inl` and `inr` for injecting into the disjunction. Those familiar with the implementation of the axiom $A \vee \neg A$ might guess that this returns the address of an A continuation, as in that case. Actually, this doesn't work! We can't build a $\Box A$ by accumulating evidence for A at different worlds. Instead, we return again a box that does something when opened.

```
letcc uo in           ($uo : \Box A \vee \Diamond\neg A \star \omega$)
  inl(box ω'.
     letcc u in
      throw(inr(get◇[ω'] here(λa. throw a to u)))
      to uo)
```

First we save the current continuation as *uo*, since we will need to "change our minds" about which clause we return! Initially, we return a box whose body also grabs the continuation (of type $A \star \omega'$) as u. Suppose the box is opened at the world ω'. We then throw to the remote continuation *uo* a program that comes back to ω', forms a term of $\neg A$, publishes it, and moves it to the first world.

To summarize, when asked for $\Box A \vee \Diamond\neg A$, the program

1. initially says $\Box A$.

2. if the box is opened, the program uses the lack of A to produce a $\Diamond\neg A$, time travels back to when it was asked about the disjunction, and returns this different answer.

3. if the $\neg A$ continuation is ever invoked, the program goes back and uses the A to fulfill the outstanding lack of A at the world where the box was opened.

In the style of sci-fi storytelling popular when describing such things, we conclude our examples with the following fable (with apologies to Wadler [16]):

> A magician who purports to be from the future is making bold claims. Asking for a volunteer, he offers the following prize to anyone who comes on stage:
>
> "I'm going to hand you a box that has *you* inside it!"
>
> "Either that, or I'll give you the address of a place with a magical time travelling portal."
>
> Being questionably brave, you volunteer and walk onto the stage. The magician hands you your prize—a large cardboard box. Noting your skepticism, he adds, "You can open it anywhere, and you'll be inside."
>
> You decide to take the box home. It's much too light to have anything in it, let alone yourself! You open the box and look inside, wondering what sort of gag he has planned. But suddenly you find that the box has disappeared, and you're standing on stage waiting for him to tell you what you've won, again.
>
> "The address of the time-travelling portal is," he begins, rattling off your home address. You are startled that he could have known your address, but when you later arrive home, you see an open cardboard box waiting. Is this supposed to be the portal? Knowing it to be harmless, but insisting upon proving the magician to be a fraud, you step into it.
>
> A hot flash of embarassment passes over you as you realize that you are now standing in a cardboard box, in your house, as promised.

5 Type System and Operational Semantics

Our deductive proof theory begets a natural programming language whose syntax is the proof terms from Figure 1. In order to give this language an operational interpretation, we need to introduce a number of syntactic constructs, which are given in Figure 3.

As in Lambda 5, the behavior of a program is specified in terms of an abstract network that steps from state to state. The network is built out of a fixed number of worlds, whose names we write as bold $\mathbf{w}$. Because we can now mention specific worlds in addition to hypothetical worlds ω, we introduce world expressions, which are written with a Roman w. A network state $\mathbb{N}$ has two parts. First is a world configuration $\mathbb{W}$ which identifies two tables

types	A, B	::=	$A \supset B \mid \Box A \mid \Diamond A \mid A \wedge B \mid \bot$
networks	$\mathbb{N}$	::=	$\mathbb{W}; R$
configs	$\mathbb{W}$	::=	$\{\mathbf{w}_1 : \langle \chi_1, b_1 \rangle, \cdots\}$
cursors	R	::=	$\mathbf{w} : [k \prec v] \mid \mathbf{w} : [k \succ M]$
tables	b	::=	$\bullet \mid b, \ell = v$
cont tables	χ	::=	$\bullet \mid \chi, \mathbf{k} = k$
config types	Σ	::=	$\{\mathbf{w}_1 : \langle X_1, \beta_1 \rangle, \cdots\}$
table types	β	::=	$\bullet \mid \beta, \ell : A$
ctable types	X	::=	$\bullet \mid X, \mathbf{k} : A$
world exps	w	::=	$\omega \mid \mathbf{w}$
world vars	ω		world names $\mathbf{w}$
labels	ℓ		value vars x, y
cont labs	$\mathbf{k}$		cont vars u
values	v	::=	$\lambda x.M \mid \texttt{box}\ \omega.M \mid \mathbf{w}.\ell \mid \langle v, v' \rangle$
conts	k	::=	$\texttt{return}\, Z \mid \texttt{finish}$
			$\mid \texttt{abort} \mid k \lhd f$
cont exps	Z	::=	$\mathbf{w}.\mathbf{k} \mid u$
frames	f	::=	$\circ\, N \mid v\ \circ \mid \texttt{here}\, \circ \mid \texttt{unbox}\, \circ$
			$\mid \texttt{letd}\, \omega.x = \circ\, \texttt{in}\, N \mid \pi_n \circ$
			$\mid \langle \circ, N \rangle \mid \langle v, \circ \rangle$
exps	M, N	::=	$v \mid M N \mid x \mid \ell \mid \texttt{get}_{\Box}[\mathrm{w}]M$
			$\mid \texttt{here}\, M \mid \texttt{get}_{\Diamond}[\mathrm{w}]M$
			$\mid \texttt{unbox}\, M \mid \texttt{letd}\, \omega.x = M\, \texttt{in}\, N$
			$\mid \texttt{rpc}[\mathrm{w}]M \mid \texttt{letcc}\, u\, \texttt{in}\, M$
			$\mid \texttt{throw}\, M\, \texttt{to}\, Z \mid \langle M, N \rangle \mid \pi_n M$

Figure 3: Syntax of type system

with each world $\mathbf{w}_i$ present. The first table χ_i stores network-wide continuations by mapping continuation labels $\mathbf{k}$ to literal continuations k. The second table b_i maps value labels ℓ to values in order to store values whose address we have published. These tables have types X and β respectively (which map labels $\mathbf{k}$ and ℓ to types), and so we can likewise construct the type of an entire configuration, written Σ.

Aside from the current world configuration, a network state also contains a *cursor* denoting the current focus of computation. The cursor either takes the form $\mathbf{w} : [k \prec v]$ (returning the value v to the continuation k) or $\mathbf{w} : [k \succ M]$ (evaluating the expression M in continuation k). In either case it selects a world $\mathbf{w}$ where the computation is taking place.

Continuations themselves are stacks of frames (expressions with a "hole," written $\circ$) with a bottommost `return`, `finish` or `abort`. The `finish` continuation represents the end of computation, so a network state whose cursor is returning a value to `finish` is called *terminal.* The `abort` continuation will be unreachable, and `return` will send the received value to a remote continuation.

Most of the expressions and values are straightforward. As in Lambda 5, the canonical value for $\Box$ abstracts over the hypothetical world and leaves its argument unevaluated ($\texttt{box}\ \omega'.M$). The canonical form for $\Diamond$ is a pair of a world name and a label $\mathbf{w}.\ell$, which addresses a table entry at that world. Such an address is well-formed anywhere (assuming that $\mathbf{w}$'s table has a label ℓ containing a value of type A) and has type $\Diamond A@\mathrm{w}'$. On the other hand we have another sort of label, written just ℓ, which is disembodied from its world. These labels arise from the `letd` construct, which deconstructs an address $\mathbf{w}.\ell$ into its components $\mathbf{w}$ and ℓ (see the $\Diamond E$ rule from Figure 1). Disembodied labels only make sense at a single world—here ℓ would have type $A\,@\,\mathbf{w}$.

Although the external language only allows a `throw` to a continuation variable, intermediate states of evaluation require that these be replaced with the continuation expression $\mathbf{w}.\mathbf{k}$, which pairs a continuation label with the world at which it lives. These continuation expressions are filled in by `letcc`.

Judgment	**Reading**
$\Sigma; \Gamma; \Delta \vdash M : A\,@\,\mathrm{w}$	The expression M has type A at world w
$\Sigma \vdash k : A\star\mathrm{w}$	The continuation k expects a value of type A at world w
$\Sigma; \Delta \vdash Z : A\star\mathrm{w}$	The continuation expression Z is well-formed with type A at w
$\Sigma \vdash b\,@\,\mathbf{w}$	The value table b is well-formed at the world named $\mathbf{w}$
$\Sigma \vdash \chi\star\mathbf{w}$	The continuation table χ is well-formed at the world named $\mathbf{w}$
$\Sigma \vdash R$	The cursor is well-formed
$\Sigma \vdash \mathbb{N}$	The network is well-formed

Figure 4: Index of Judgments. In each judgment Σ is a configuration typing, Γ is a context of truth hypotheses, and Δ is a context of falsehood hypotheses

The type system is given in Figure 5 (we omit for space the rules that are the same as in Figure 1 except for the configuration typing Σ). The index of judgments in Figure 4 may be a useful reference in understanding them.

The rules *addr* and *lab* are used to type run-time artifacts of address publishing. In either case, we look up the type in the appropriate table typing β. As mentioned, *throw* allows a continuation expression Z, which must take the form of a variable (typed with *hyp**, as in the logic) or address into a continuation table.

Typing of literal continuations k is fairly unsurprising. Note that the judgment $\Sigma \vdash k : A\star\mathrm{w}$ means that the continuation k *expects* a value of type A at w. The `return` continuation arises only from a $\texttt{get}_\Diamond$ or $\texttt{get}_\Box$, and so it allows only values of type $\Diamond A$ or $\Box A$. We re-use the network continuation mechanism here to refer to the outstanding $\texttt{get}_\Diamond$ or $\texttt{get}_\Box$ on the remote machine.

For an entire network to be well-formed (rule *net*), all of the tables must have the type indicated by the configuration type Σ, which means that they must have exactly the same

$$\dfrac{\Sigma(\mathbf{w}) = \langle X, \beta\rangle \quad \beta(\ell) = A}{\Sigma; \Gamma; \Delta \vdash \mathbf{w}.\ell : \Diamond A @ \mathrm{w}'}\ \text{addr} \qquad \dfrac{\Sigma(\mathbf{w}) = \langle X, \beta\rangle \quad \beta(\ell) = A}{\Sigma; \Gamma; \Delta \vdash \ell : A @ \mathbf{w}}\ \text{lab}$$

$$\dfrac{\Sigma; \Gamma; \Delta \vdash M : A @ \mathrm{w} \quad \Sigma; \Delta \vdash Z : A \star \mathrm{w}}{\Sigma; \Gamma; \Delta \vdash \texttt{throw}\, M\, \texttt{to}\, Z : C @ \mathrm{w}'}\ \text{throw} \qquad \dfrac{\Sigma; \Gamma; \Delta, u : A \star \mathrm{w} \vdash M : A @ \mathrm{w}}{\Sigma; \Gamma; \Delta \vdash \texttt{letcc}\, u\, \texttt{in}\, M : A @ \mathrm{w}}\ \text{letcc}$$

$$\dfrac{\Sigma(\mathbf{w}) = \langle X, \beta\rangle \quad X(\mathbf{k}) = A}{\Sigma; \Delta \vdash \mathbf{w}.\mathbf{k} : A \star \mathbf{w}}\ \text{addr}^{\star} \qquad \dfrac{}{\Sigma; \Delta, u : A \star \mathrm{w} \vdash u : A \star \mathrm{w}}\ \text{hyp}^{\star}$$

$$\dfrac{}{\Sigma \vdash \texttt{abort} : \bot \star \mathrm{w}}\ \text{kabort} \qquad \dfrac{}{\Sigma \vdash \texttt{finish} : A \star \mathrm{w}}\ \text{kfinish}$$

$$\dfrac{A = \Box A' \text{ or } \Diamond A' \quad \Sigma; \cdot \vdash Z : A \star \mathrm{w}'}{\Sigma \vdash \texttt{return}\, Z : A \star \mathrm{w}}\ \text{kret} \qquad \dfrac{\Sigma \vdash k : \Diamond A \star \mathrm{w}}{\Sigma \vdash k \lhd \texttt{here}\, \circ : A \star \mathrm{w}}\ \text{khere}$$

$$\dfrac{\Sigma \vdash k : B \star \mathrm{w} \quad \Sigma; \cdot; \cdot \vdash N : A @ \mathrm{w}}{\Sigma \vdash k \lhd \circ N : A \supset B \star \mathrm{w}}\ \text{kapp}_1 \qquad \dfrac{\Sigma \vdash k : B \star \mathrm{w} \quad \Sigma; \cdot; \cdot \vdash v : A \supset B @ \mathrm{w}}{\Sigma \vdash k \lhd v \circ : A \star \mathrm{w}}\ \text{kapp}_2$$

$$\dfrac{\Sigma \vdash k : C \star \mathrm{w} \quad \Sigma; x : A @ \omega; \overset{\overset{\omega}{\vdots}}{\cdot} \vdash N : C @ \mathrm{w}}{\Sigma \vdash k \lhd \texttt{letd}\, \omega.x = \circ\, \texttt{in}\, N : \Diamond A \star \mathrm{w}}\ \text{kletd} \qquad \dfrac{\Sigma \vdash k : A \star \mathrm{w}}{\Sigma \vdash k \lhd \texttt{unbox}\, \circ : \Box A \star \mathrm{w}}\ \text{kunbox}$$

$$\dfrac{\Sigma \vdash k : A \wedge B \star \mathrm{w} \quad \Sigma; \cdot; \cdot \vdash N : B @ \mathrm{w}}{\Sigma \vdash k \lhd \langle \circ, N\rangle : A \star \mathrm{w}}\ \text{kpair}_1 \qquad \dfrac{\Sigma \vdash k : A \wedge B \star \mathrm{w} \quad \Sigma; \cdot; \cdot \vdash v : A @ \mathrm{w}}{\Sigma \vdash k \lhd \langle v, \circ\rangle : B \star \mathrm{w}}\ \text{kpair}_2$$

$$\dfrac{\beta = (\ell_1 : A_1, \dots) \quad \Sigma; \cdot; \cdot \vdash v_1 : A_1 @ \mathbf{w} \quad \dots}{\underbrace{\{\cdots, \mathbf{w} : \langle X, \beta\rangle, \cdots\}}_{\Sigma} \vdash \underbrace{\ell_1 = v_1, \dots}_{b} @ \mathbf{w}}\ b \qquad \dfrac{X = (\mathbf{k}_1 : A_1, \dots) \quad \Sigma \vdash k_1 : A_1 \star \mathbf{w} \quad \dots}{\underbrace{\{\cdots, \mathbf{w} : \langle X, \beta\rangle, \cdots\}}_{\Sigma} \vdash \underbrace{\mathbf{k}_1 = k_1, \dots}_{\chi} \star \mathbf{w}}\ \chi$$

$$\dfrac{\begin{array}{l}\mathbf{w} \in \mathrm{dom}(\Sigma) \\ \Sigma; \cdot; \cdot \vdash v : A @ \mathbf{w} \quad \Sigma \vdash k : A \star \mathbf{w}\end{array}}{\Sigma \vdash \mathbf{w} : [k \prec v]}\ \text{ret} \qquad \dfrac{\begin{array}{l}\mathbf{w} \in \mathrm{dom}(\Sigma) \\ \Sigma; \cdot; \cdot \vdash M : A @ \mathbf{w} \quad \Sigma \vdash k : A \star \mathbf{w}\end{array}}{\Sigma \vdash \mathbf{w} : [k \succ M]}\ \text{eval}$$

$$\dfrac{\Sigma \vdash R \quad \Sigma \vdash \chi_i @ \mathbf{w}_i \ \dots \quad \Sigma \vdash b_i @ \mathbf{w}_i \ \dots}{\Sigma \vdash \{\mathbf{w}_1 : \langle \chi_1, b_1\rangle, \cdots, \mathbf{w}_m : \langle \chi_m, b_m\rangle\}; R}\ \text{net}$$

Figure 5: Type System

labels, and the values or continuations must be well-typed at the specified types (rules b and χ). Finally, the cursor must be well-formed: it must select a world that exists in the network, and there must exist a type A such that its continuation and value or expression both have type A and are closed.

Having set up the syntax and type system, we can now give the operational semantics and type safety theorem. After the following section we remark on how the semantics can be made concurrent, and some thoughts on applications of distributed continuations.

$\supset_e$-p	$\mathbb{W}; \mathbf{w} : [k \succ M N]$	$\mapsto$	$\mathbb{W}; \mathbf{w} : [k \lhd \circ N \succ M]$
$\supset_e$-s	$\mathbb{W}; \mathbf{w} : [k \lhd \circ N \prec v]$	$\mapsto$	$\mathbb{W}; \mathbf{w} : [k \lhd v \circ \succ N]$
$\supset_e$-r	$\mathbb{W}; \mathbf{w} : [k \lhd (\lambda x.M)\circ \prec v]$	$\mapsto$	$\mathbb{W}; \mathbf{w} : [k \succ [v/x]M]$
$\wedge_i$-p	$\mathbb{W}; \mathbf{w} : [k \succ \langle M, N \rangle]$	$\mapsto$	$\mathbb{W}; \mathbf{w} : [k \lhd \langle \circ, N \rangle \succ M]$
$\wedge_i$-s	$\mathbb{W}; \mathbf{w} : [k \lhd \langle \circ, N \rangle \prec v]$	$\mapsto$	$\mathbb{W}; \mathbf{w} : [k \lhd \langle v, \circ \rangle \succ N]$
$\wedge_i$-r	$\mathbb{W}; \mathbf{w} : [k \lhd \langle v_1, \circ \rangle \prec v_2]$	$\mapsto$	$\mathbb{W}; \mathbf{w} : [k \prec \langle v_1, v_2 \rangle]$
$\wedge_{e_n}$-p	$\mathbb{W}; \mathbf{w} : [k \succ \pi_n M]$	$\mapsto$	$\mathbb{W}; \mathbf{w} : [k \lhd \pi_n \circ \succ M]$
$\wedge_{e_n}$-r	$\mathbb{W}; \mathbf{w} : [k \lhd \pi_n \prec \langle v_1, v_2 \rangle]$	$\mapsto$	$\mathbb{W}; \mathbf{w} : [k \prec v_n]$
$\Box_i$-v	$\mathbb{W}; \mathbf{w} : [k \succ \texttt{box}\ \omega.M]$	$\mapsto$	$\mathbb{W}; \mathbf{w} : [k \prec \texttt{box}\ \omega.M]$
$\Diamond_i$-v	$\mathbb{W}; \mathbf{w} : [k \succ \mathbf{w}'.\ell]$	$\mapsto$	$\mathbb{W}; \mathbf{w} : [k \prec \mathbf{w}'.\ell]$
$\supset_i$-v	$\mathbb{W}; \mathbf{w} : [k \succ \lambda x.M]$	$\mapsto$	$\mathbb{W}; \mathbf{w} : [k \prec \lambda x.M]$
$\Diamond_i$-p	$\mathbb{W}; \mathbf{w} : [k \succ \texttt{here}\, M]$	$\mapsto$	$\mathbb{W}; \mathbf{w} : [k \lhd \texttt{here} \circ \succ M]$
$\Diamond_i$-r	$\{\mathbf{w} : \langle \chi, b \rangle, \cdots\}; \mathbf{w} : [k \lhd \texttt{here} \circ \prec v]$	$\mapsto$	$\{\mathbf{w} : \langle \chi, (b, \ell = v) \rangle, \cdots\}; \mathbf{w} : [k \prec \mathbf{w}.\ell]$
			(ℓ fresh)
ℓ-r	$\{\mathbf{w} : \langle \chi, b \rangle, \cdots\}; \mathbf{w} : [k \succ \ell]$	$\mapsto$	$\{\mathbf{w} : \langle \chi, b \rangle, \cdots\}; \mathbf{w} : [k \prec v]$
			$(b(\ell) = v)$
$\Diamond_e$-p	$\mathbb{W}; \mathbf{w} : [k \succ \texttt{letd}\, \omega.x = M\, \texttt{in}\, N]$	$\mapsto$	$\mathbb{W}; \mathbf{w} : [k \lhd \texttt{letd}\, \omega.x = \circ\, \texttt{in}\, N \succ M]$
$\Diamond_e$-r	$\mathbb{W}; \mathbf{w} : [k \lhd \texttt{letd}\, \omega.x = \circ\, \texttt{in}\, N \prec \mathbf{w}'.\ell]$	$\mapsto$	$\mathbb{W}; \mathbf{w} : [k \succ [\ell/x][\mathbf{w}'/\omega]N]$
$\Box_e$-p	$\mathbb{W}; \mathbf{w} : [k \succ \texttt{unbox}\, M]$	$\mapsto$	$\mathbb{W}; \mathbf{w} : [k \lhd \texttt{unbox} \circ \succ M]$
$\Box_e$-r	$\mathbb{W}; \mathbf{w} : [k \lhd \texttt{unbox} \circ \prec \texttt{box}\ \omega.M]$	$\mapsto$	$\mathbb{W}; \mathbf{w} : [k \succ [\mathbf{w}/\omega]M]$
letcc	$\{\mathbf{w} : \langle \chi, b \rangle, \cdots\}; \mathbf{w} : [k \succ \texttt{letcc}\, u\, \texttt{in}\, M]$	$\mapsto$	$\{\mathbf{w} : \langle (\chi, \mathbf{k} = k), b \rangle, \cdots\}; \mathbf{w} : [k \succ [\mathbf{w}.\mathbf{k}/u]M]$
			($\mathbf{k}$ fresh)
throw	$\{\mathbf{w}' : \langle \chi, b \rangle, \cdots\}; \mathbf{w} : [k \succ \texttt{throw}\, M\, \texttt{to}\, \mathbf{w}'.\mathbf{k}]$	$\mapsto$	$\{\mathbf{w}' : \langle \chi, b \rangle, \cdots\}; \mathbf{w}' : [k' \succ M]$
			$(\chi(\mathbf{k}) = k')$
rpc	$\mathbb{W}; \mathbf{w} : [k \succ \mathit{rpc}[\mathbf{w}']M]$	$\mapsto$	$\mathbb{W}; \mathbf{w}' : [\texttt{abort} \succ M]$
			$(\mathbf{w} \in \mathrm{dom}(\mathbb{W}))$
$\Box_m$	$\{\mathbf{w} : \langle \chi, b \rangle, \cdots\}; \mathbf{w} : [k \succ \texttt{get}_{\Diamond}[\mathbf{w}']M]$	$\mapsto$	$\{\mathbf{w} : \langle (\chi, \mathbf{k} = k), b \rangle, \cdots\}; \mathbf{w}' : [\texttt{return}\, \mathbf{w}.\mathbf{k} \succ M]$
			($\mathbf{k}$ fresh)
$\Diamond_m$	$\{\mathbf{w} : \langle \chi, b \rangle, \cdots\}; \mathbf{w} : [k \succ \texttt{get}_{\Box}[\mathbf{w}']M]$	$\mapsto$	$\{\mathbf{w} : \langle (\chi, \mathbf{k} = k), b \rangle, \cdots\}; \mathbf{w}' : [\texttt{return}\, \mathbf{w}.\mathbf{k} \succ M]$
			($\mathbf{k}$ fresh)
ret	$\{\mathbf{w} : \langle \chi, b \rangle, \cdots\}; \mathbf{w}' : [\texttt{return}\, \mathbf{w}.\mathbf{k} \prec v]$	$\mapsto$	$\{\mathbf{w} : \langle \chi, b \rangle, \cdots\}; \mathbf{w} : [k \prec v]$
			$(\chi(\mathbf{k}) = k)$

Figure 6: Operational Semantics

5.1 Operational Semantics

The operational semantics of our language is given in Figure 6, as a binary relation $\mapsto$ between network states. The semantics evaluates programs sequentially, though we give a concurrent semantics in Section 6.

As should be obvious, the semantics is continuation-based. At any step, the cursor is selecting a world and continuation, with a value to return to it or an expression to evaluate. The rules generally fall into a few categories, as exemplified by the (standard) rules for $\supset$: There are **p**ush rules, in which we begin evaluating a subexpression of some M, pushing the context into the continuation, **s**wap rules, where we have finished evaluating one subexpression and move onto the next, and **r**eduction rules, where we have a value and actually do something with it. Every well-typed machine state will be closed with respect to truth, falsehood, and world hypotheses, so we don't have rules for variables and can specialize some rules.

The first interesting rule is $\Diamond_i$-r. It publishes the value v and returns its address by generating a new label, mapping that label to v within its value table, and returning the pair $\mathbf{w}.\ell$, where $\mathbf{w}$ is the current world. Whenever we try to evaluate a label (rule ℓ-r), we look it up in the current world's value table in order to fetch the value. A key consequence of type safety (Theorems 5, 6) is that labels are only evaluated in the correct world. To eliminate an address (rule $\Diamond_e$-r) we substitute the constituent world and label through the body of the `letd`. Note that this step is slightly non-standard, because we substitute the *expression* ℓ for a variable rather than some value. But because the variable is in general at a different world, we are not in a position to get its value yet. We instead wait until the expression ℓ is sent to its home world (perhaps as part of some larger expression) to be looked up. The rules for $\Box$ are much simpler: `box` $\omega.M$ is already a value (rule $\Box_i$-v), and to `unbox` we simply substitute the current world for the hypothetical one (rule $\Box_e$-r).

When encountering a `letcc`, we grab the current continuation k. Because the continuation may be referred to from elsewhere in the network, we publish it in a table and form a global address for it (of the form $\mathbf{w}.\mathbf{k}$), just as we did for $\Diamond$ addresses. This value is substituted for the falsehood variable u using standard substitution—*not* the special falsehood substitition we used in Section 2. The latter was a proof-theoretic notion used to eliminate uses of the hypothesis; here we want the use of the hypothesis (`throw`) to have run-time significance. A point of comparison is the above paragraph, where we substituted the expression ℓ for a variable because we wanted to delay the operation until the time the variable is "looked up."

Throwing to a continuation (rule *throw*) is handled straightforwardly. The continuation expression will be closed, and therefore of the form $\mathbf{w}'.\mathbf{k}$. We look up the label $\mathbf{k}$ in $\mathbf{w}'$—or rather, *cause* $\mathbf{w}'$ to look it up—and pass the expression M to it. Note that we do not evaluate the argument before throwing it to the remote continuation. In general we *can not* evaluate it, because it is only well-typed at the remote world, which may be different from the world we're in.

Finally, we have the rules that move between worlds. The rule for `rpc` is easiest; since the target world expression must be closed it will be a world constant in the domain of $\mathbb{W}$. We simply move the cursor to that world (destroying the current continuation, which can never be reached), and begin evaluating the expression M under the unreachable continuation `abort`. The rules for $\texttt{get}_\Diamond$ and $\texttt{get}_\Box$ work similarly, but they need to save the current continuation since they will be returned to! These steps push a `return` frame, which reduces like `throw`. In contrast, however, the argument (of type $\Box A$ or $\Diamond A$) will be eagerly evaluated, because such values are portable. (After all, the whole point is to create the box at one world and then move it to another.)

In order for our language to make sense it must be type safe; any well-typed program must have a well-defined meaning as a sequence of steps in the abstract network. Type safety is stated as usual in terms of progress and preservation:

Theorem 5 (Progress)
If $\Sigma \vdash \mathbb{N}$
then *either* $\mathbb{N}$ *is terminal* *or* $\exists \mathbb{N}'.\mathbb{N} \mapsto \mathbb{N}'$.

Theorem 6 (Preservation)
If $\quad \Sigma \vdash \mathbb{N}$ *and* $\mathbb{N} \mapsto \mathbb{N}'$
then $\quad \exists \Sigma'.\ \Sigma' \supseteq \Sigma$ *and* $\Sigma' \vdash \mathbb{N}'$.

Progress says that any well-formed network state can take another step, or is done. (Recall a *terminal* network is one where the cursor is returning a value to a `finish` continuation.) Preservation says that any well-typed network state that takes a step results in another well-typed state (perhaps in an extended configuration typing Σ'[9]). By iterating alternate applications of these theorems we see that any well-typed program is able to step repeatedly and remain well-formed, or else eventually comes to rest in a terminal state.

6 Concurrency and Communication

Many distributed computing problems benefit from concurrency, with one or more processes running on each node in the network. This section gives some brief thoughts on concurrency in our classical calculus.

First-class continuations are often used in the implementation of coroutines. With primitives for recursion and state we could also implement coroutines in C5, however, such an implementation is silly because it would require the implementation of a global scheduler, and would anyway defeat the purpose of concurrency on multiple nodes—only one coroutine would be running at any given time!

Fortunately, our operational semantics admits ad hoc concurrency easily. If we simply replace the cursor R in our network state "$\mathbb{W}; R$" with a multiset of cursors $\mathfrak{R}$, then we can permit a step on any one of these cursors essentially according to the old rules. Formally,

$$\begin{array}{ll} & \mathbb{W}; \mathfrak{R} \quad \mapsto^c \quad \mathbb{W}'; \mathfrak{R}' \\ \text{iff} & \mathfrak{R} = R \uplus \mathfrak{R}_{\text{rest}} \\ \text{and} & \mathbb{W}; R \mapsto \mathbb{W}'; R' \\ \text{and} & \mathfrak{R}' = R' \uplus \mathfrak{R}_{\text{rest}} \end{array}$$

We can then add primitives as desired to spawn new cursors. A very simple one evaluates M and N in parallel and returns each one to the same continuation.

$$\frac{\Gamma; \Delta \vdash M : A@\mathrm{w} \quad \Gamma; \Delta \vdash N : A@\mathrm{w}}{\Gamma; \Delta \vdash M|N : A@\mathrm{w}}\ \text{par}$$

$$\mathbb{W}; \mathfrak{R} \uplus \mathbf{w}{:}[k \succ M|N] \mapsto^c \mathbb{W}; \mathfrak{R} \uplus \mathbf{w}{:}[k \succ M] \uplus \mathbf{w}{:}[k \succ N]$$

A suitable extension of type safety holds for $\mapsto^c$.

With concurrency in place we can implement CML-style channels [13] with the help of continuations (and a few other features for developing mutable recursive structures). The type of a channel carrying values of type A could be:

$$A\ \texttt{chan} \quad \doteq \quad \Diamond(A\ \texttt{queue} \wedge (\neg A)\ \texttt{queue})$$

[9] $\Sigma' \supseteq \Sigma$ iff Σ' and Σ each describe the same set of worlds, and for each world, if $X'(\mathbf{k}) = A$ then $X(\mathbf{k}) = A$, and likewise for β' and β.

Here a channel is represented as the address of a pair of queues. In order to send to this channel, the sender must be able to bring a value of type A to the world where the channel lives, so it must be a box or diamond type itself (or see Section 8 for more options). The first queue holds the values that have been sent on the channel and not yet received, the second holds the continuations of outstanding `recvs`. To implement `recv` (assuming no values are waiting in the first queue), we grab the current continuation, enqueue it, and abort.

This is a standard technique; the point here is to emphasize the utility of continuations as primitives for implementing useful distributed computing features.

7 Disjunction

To add disjunction to C5, we need to use the following elimination form in order to preserve the correspondence with classical S5:

$$\frac{\begin{array}{c}\Gamma;\Delta\vdash M : A\vee B\,@\,\omega' \\ \Gamma, x{:}A\,@\,\omega';\Delta\vdash N_1 : C\,@\,\omega \\ \Gamma, x{:}B\,@\,\omega';\Delta\vdash N_2 : C\,@\,\omega\end{array}}{\Gamma;\Delta\vdash \begin{array}{l}\texttt{case}\,M\,\texttt{of} \\ \quad\texttt{inl}\,x\Rightarrow N_1 \\ \mid\ \texttt{inr}\,x\Rightarrow N_2\end{array} : C\,@\,\omega}\ \vee E$$

This rule is completely unsurprising except that the case object M is at a *different world*, ω'. In our logic we've tried hard to avoid this sort of *action-at-a-distance*, instead preferring to have our introduction and elimination rules compute locally. However, a motion rule for disjunction is out of the question, because it is unsound: it is not the case that if $\Gamma;\Delta\vdash A\vee B\,@\,\omega$ then necessarily $\Gamma;\Delta\vdash A\vee B\,@\,\omega'$. In our previous paper we speculated that this rule could be implemented nonetheless by sending back merely a *bit* telling the case-analyzing world which branch it should enter, but this requires some suspicious operational machinery. The same is true in the classical case, which is why we have avoided treating disjunction so far.

The problem with a local rule (where all four worlds are the same) comes when translating the sequent calculus rule

$$\frac{\begin{array}{c}\Gamma, A\vee B\,@\,\omega, A\,@\,\omega\ \#\ \Delta \\ \Gamma, A\vee B\,@\,\omega, B\,@\,\omega\ \#\ \Delta\end{array}}{\Gamma, A\vee B\,@\,\omega\ \#\ \Delta}\ \vee T$$

into natural deduction for Theorem 4(b). We have by induction

$$\begin{array}{c}\Gamma, x{:}A\vee B\,@\,\omega, a{:}A\,@\,\omega;\Delta\vdash N_1 : C\,@\,\omega'\ (\forall C,\omega') \\ \Gamma, x{:}A\vee B\,@\,\omega, b{:}B\,@\,\omega;\Delta\vdash N_2 : C\,@\,\omega'\ (\forall C,\omega')\end{array}$$

and a derivation of $\Gamma, x{:}A\vee B\,@\,\omega;\Delta\vdash x : A\vee B\,@\,\omega$ by the *hyp* rule. However, we cannot apply a local $\vee$-elimination rule, because it would require its object x to be at the same world as its conclusion. Thus we are able to prove C at ω, but not at *all* worlds.

As it turns out, support for disjunction and remote disjunction elimination is already present in C5, thanks to one of de Morgan's laws. Suppose that we define $A \vee B$ as follows

$$A \vee B \doteq \neg(\neg A \wedge \neg B)$$

That is, $A \vee B$ becomes a continuation that takes *two* continuations: one if the disjunct is A, and one if the disjunct is B. This technique is well-known for CPS conversion, and first-class continuations let us do it without having to CPS-convert the entire program. Encoding the injections is easy:

$$\begin{array}{lcl} \mathtt{inl}\, M & \doteq & \lambda ab.(\pi_1 ab)M \\ \mathtt{inr}\, M & \doteq & \lambda ab.(\pi_2 ab)M \end{array}$$

Encoding local case analysis is standard:

$$\begin{array}{lcl} \mathtt{case}\, M\, \mathtt{of} & & \mathtt{letcc}\, u \\ \quad \mathtt{inl}\, x \Rightarrow N_1 & \doteq & \mathtt{in}\, M\langle \lambda x.\, \mathtt{throw}\, N_1\, \mathtt{to}\, u, \\ \mid\ \mathtt{inr}\, x \Rightarrow N_2 & & \qquad \lambda x.\, \mathtt{throw}\, N_2\, \mathtt{to}\, u\rangle \end{array}$$

Finally, because we have locally CPS-converted, we can do the case analysis remotely, and rely on `throw` to get us back:

$$\begin{array}{lcl} \mathtt{case}\, M\, \mathtt{of} & & \mathtt{letcc}\, u \\ \quad \mathtt{inl}\, x \Rightarrow N_1 & \doteq & \mathtt{in}\, \mathtt{rpc}[\omega']M\langle \lambda x.\, \mathtt{throw}\, N_1\, \mathtt{to}\, u, \\ \mid\ \mathtt{inr}\, x \Rightarrow N_2 & & \qquad\qquad \lambda x.\, \mathtt{throw}\, N_2\, \mathtt{to}\, u\rangle \end{array}$$

This has exactly the same typing conditions as the remote rule above; x is bound to the remote type $A @ \omega'$, even though the expression N_1 is evaluated at ω.

Classical logic is ripe with possibilities for definition. It is interesting to consider their implications. Recall that in Section 4 we proved $\Diamond A$ equivalent to $\neg\Box\neg A$. This means that, as classicists typically do, we could then just consider $\Diamond A$ as a derived form. This would amount to a roundabout way of using the continuation table to publish values rather than the value table. Clearly, we could also take the even stranger route of defining $\Box A$ in terms of $\Diamond$, which gives us a mobile code "server" that sends code to our continuation whenever we like.

8 Generalizing get

The typing rules and operational semantics for $\mathtt{get}_\Diamond$ and $\mathtt{get}_\Box$ are almost identical, suggesting the possibility of factoring out the common functionality into a single construct. In fact, when we add base types to C5 the desire for such a general `get` mechanism becomes clear—the calculus does not currently support a way to directly retrieve simple data like integers, so the programmer is required to implement this mobility himself by deconstructing the value at the source world and reintroducing it at the destination.

Not all types support this kind of mobility. Therefore, we create a judgment $A\,\mathsf{mobile}$, and only allow `get` on types that satisfy it.

$$\frac{\Gamma;\Delta \vdash M : A@\omega' \quad A\,\mathsf{mobile}}{\Gamma;\Delta \vdash \mathtt{get}[\omega']M : A@\omega}\ \mathtt{get}$$

The mobile judgment is defined inductively:

$$\frac{}{\Box A\,\mathsf{mobile}} \qquad \frac{}{\Diamond A\,\mathsf{mobile}}$$

$$\frac{A\,\mathsf{mobile} \quad B\,\mathsf{mobile}}{A \wedge B\,\mathsf{mobile}} \qquad \frac{}{\mathtt{int}\,\mathsf{mobile}}$$

$$\frac{}{\bot\,\mathsf{mobile}}$$

Note that $\Box A$ is always mobile, whereas $A \wedge B$ is only mobile if both of its constituent types are also mobile. We have also left out implication entirely. The general principle that decides mobility for a type is as follows: Type A can be considered mobile if whenever $\Gamma;\Delta \vdash A@\omega$, then $\Gamma;\Delta \vdash A@\omega'$ for any ω'. This tells us when we can logically introduce a `get` at A without getting in trouble. However, because `get` is only required for completeness at $\Box$ and $\Diamond$ types (since we have otherwise reduced their strength), we have significant flexibility in what other types we allow to be mobile. This is largely an operational concern, the question being: Can we provide a more efficient implementation than the one that the user would otherwise have to write? Sometimes the answer is clearly yes, as in the case of integers. For other types the answer is probably no; although if A and B are mobile then $A \supset B$ satisfies the principle above, implementing `get` primitively for functions seems to require building a proxy for the function, which is what a programmer would do to implement this mobility himself.

9 Conclusions

9.1 Related Work

Parigot's $\lambda\mu$-calculus has inspired many computational proof systems for classical logic, including Wadler's dual calculus [16]. The calculus is sequent-oriented and contains *cut* as a computational primitive, emphasizing the duality of computing with values and covalues (continuations). For programming in C5, we choose a natural deduction system which is deliberately *non*-dual. We bias the logic towards truth, which corresponds to computing mainly with values (as is typical) rather than covalues. Nevertheless, we expect that a dual version of classical S5 could be easily made to work, perhaps starting from the sequent calculus presented in Section 3.

Recently, others have used modal logic to describe distributed tasks or as the basis for programming languages, although we know of no modal systems that feature distributed continuations.

Borghuis and Feijs give an early computational interpretation of modal logic in their intuitionistic Modal Type System for Networks [2]. Their calculus and logic describe programs on a network with stationary services (e.g., printers, file converters) and mobile data (e.g., documents). They use $\Box$, annotated with a location, to represent services. For example, $\Box^o(A \supset B)$ means a function from A to B at the location o. However, the calculus has no way of internalizing mobility as a proposition, so mobile data is limited to base types. Services are similarly restricted to depth-one arrow types. By using $\Box$ for mobile code and $\Diamond$ for stationary resources, we believe our resulting calculus is both simpler and more general.

Moody [7] gives a system based on the constructive modal logic S4 due to Pfenning and Davies [11]. This language is based on judgments $A\,\mathsf{true}$ (here), $A\,\mathsf{poss}$ (somewhere), and $A\,\mathsf{valid}$ (everywhere) rather than the "explicit worlds" formulation of Lambda 5 and C5. The operational semantics of his system takes the form of a process calculus with nondeterminism, concurrency and synchronization; a significantly different approach from our sequential abstract machine. Interpreted in the Kripke model, S4's accessibility relation satisfies only reflexivity and transitivity, not symmetry. Moody uses the limited accessibility to express process interdependence rather than—as we do—connections between actual network locations. Programs are therefore somewhat higher-level and express *potential mobility* instead of explicit code motion as in our mobility rules. In particular, due to the lack of symmetry it is not possible to return to world after a remote procedure call from it, except by returning a value.

Jia and Walker [6] give a judgmental account of an S5-like system based on intuitionistic hybrid logic. Hybrid logics internalize worlds inside propositions by including a *proposition* that a value of type A resides at world ω, "$A\,\mathtt{at}\,\omega$." This leads to a technically different logic and language although they give a similar interpretation to the modalities. Their rules for $\Box$ and $\Diamond$ are non-local, which means that they rely heavily on global actions. Like Moody, they give their network semantics as a process calculus with passive synchronization across processes as a primitive notion.

9.2 Future Work

Our language now has a full arsenal of connectives and control operators, each connected to logic. Much work remains before C5 can be a practical programming language rather than exploratory calculus. Some are routine—adding extra-logical primitives like recursion and references—and some are difficult—compilation of mobile code fragments, distributed garbage collection, failure recovery, and certification.

Although we believe that C5 accomodates concurrency easily, it would be nice to have a logically-inspired account of it. Some other directions remain open to try. Proof search in linear logic sequent calculus [4] is known to admit an interpretation as concurrent computation [3]. Perhaps linear S5 in sequent style would be able to elegantly express both spatial properties and concurrency in logic?

We have presented a proof theory and corresponding programming language, C5, based on the classical modal logic S5. By exploiting the modalities we are able to give a logical account of mobility and locality, and thus an expressive programming language for distributed computing. From the logic's classical nature we derive the mechanism of distributed continuations, which creates a new connection between the $\Box$ and $\Diamond$ connectives, and forms a basis for the implementation of distributed computing primitives.

Acknowledgements

Thanks to Jason Reed and Frank Pfenning for their suggestions and expertise with LF encodings.

References

[1] Andreas Abel. A third-order representation of the $\lambda\mu$-Calculus. In S.J. Ambler, R.L. Crole, and A. Momigliano, editors, *Electronic Notes in Theoretical Computer Science*, volume 58. Elsevier, 2001.

[2] Tijn Borghuis and Loe M. G. Feijs. A constructive logic for services and information flow in computer networks. *The Computer Journal*, 43(4):274–289, 2000.

[3] Jean-Yves Girard. Towards a geometry of interaction. *Contemporary Mathematics*, 92:69–108.

[4] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–102, January 1987.

[5] Timothy G. Griffin. The formulae-as-types notion of control. In *Conf. Record 17th Annual ACM Symp. on Principles of Programming Languages, POPL'90, San Francisco, CA, USA, 17–19 Jan 1990*, pages 47–57. ACM Press, New York, 1990.

[6] Limin Jia and David Walker. Modal proofs as distributed programs. *13th European Symposium on Programming*, pages 219–223, March 2004.

[7] Jonathan Moody. Modal logic as a basis for distributed computation. Technical Report CMU-CS-03-194, Carnegie Mellon University, Oct 2003.

[8] Tom Murphy, VII, Karl Crary, Robert Harper, and Frank Pfenning. A symmetric modal lambda calculus for distributed computing. In *Proceedings of the 19th IEEE Symposium on Logic in Computer Science (LICS 2004)*. IEEE Press, July 2004.

[9] Chetan Murthy. Classical proofs as programs: How, what and why. Technical Report TR91-1215, Cornell University, 1991.

[10] Michel Parigot. $\lambda\mu$-Calculus: An algorithmic interpretation of classical natural deduction. In Andrei Voronkov, editor, *Logic Programming and Automated Reasoning, International Conference LPAR'92, St. Petersburg, Russia, July 15-20, 1992, Proceedings*, volume 624 of *Lecture Notes in Computer Science*. Springer, 1992.

[11] Frank Pfenning and Rowan Davies. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science*, 11:511–540, 2001. Notes to an invited talk at the *Workshop on Intuitionistic Modal Logics and Applications* (IMLA'99), Trento, Italy, July 1999.

[12] Frank Pfenning and Carsten Schürmann. System description: Twelf – a meta-logical framework for deductive systems. In Harald Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction*, pages 202–206, Trento, Italy, July 1999. Springer-Verlag. LNAI 1632.

[13] John H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, Cambridge, England, 1999.

[14] Carsten Schürmann and Frank Pfenning. A coverage checking algorithm for LF. In D. Basin and B. Wolff, editors, *Proceedings of the 16th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2003)*, pages 120–135, Rome, Italy, September 2003. Springer-Verlag LNCS 2758.

[15] Alex Simpson. *The Proof Theory and Semantics of Intuitionistic Modal Logic.* PhD thesis, University of Edinburgh, 1994.

[16] Philip Wadler. Call-by-value is dual to call-by-name. In *Proceedings of the 8th International Conference on Functional Programming (ICFP).* ACM Press, August 2003.

APPENDIX

A Mechanized Proofs

This appendix contains Twelf [12] code for the substitutions (which are free in LF except for falsehood) and Theorems 3 (excluded middle) and 4 (equivalence). The proof of type safety is considerably longer and less nice, because it must deal with stores and store types, which must be implemented manually in LF. It is omitted from this report for space, but can be found online at `http://www.cs.cmu.edu/̃concert/`.

```
%% Classical Judgmental S5
%% Tom Murphy VII
%% Thanks: Frank Pfenning and Jason Reed
%%  8 Apr 2004
world : type.                               %name world W w.
prop : type.                                %name prop A.

%%% Natural deduction

% truth assumptions and conclusions
@ : prop -> world -> type.                  %name @ N.
%infix none 1 @.
% falsehood (continuation) assumptions
* : prop -> world -> type.                  %name * X.
%infix none 1 *.

% Implication
=> : prop -> prop -> prop.
%infix right 8 =>.
=>I : (A @ W -> B @ W) -> (A => B @ W).
=>E : (A => B @ W) -> A @ W -> B @ W.

% Necessity
□ : prop -> prop.
%prefix 9 □.
□I : (o:world A @ o) -> □ A @ W.
□E : □ A @ W -> A @ W.
□G : □ A @ W' -> □ A @ W.
```

```
% Possibility
◇ : prop -> prop.
%prefix 9 ◇.
◇I : A @ W -> ◇ A @ W.
◇E : ◇ A @ W -> (o:world A @ o -> C @ W) -> C @ W.
◇G : ◇ A @ W' -> ◇ A @ W.

% Conjunction
& : prop -> prop -> prop.
%infix none 9 &.
&I : A @ W -> B @ W -> (A & B @ W).
&E1 : (A & B @ W) -> A @ W.
&E2 : (A & B @ W) -> B @ W.

% Falsehood
⊥ : prop.
⊥E : ⊥ @ W -> C @ W'.

% Structural Rules

letcc : (A * W -> A @ W) ->
             A @ W.

throw :     A @ W ->
        (A * W -> C @ W').

%%% Sequent calculus

true  : prop -> world -> type.       %name true T t.
false : prop -> world -> type.       %name false F f.

# : type.

% judgmental
contra : true A W -> false A W -> #.

% arrow
=>T :
      (false A W -> #)    ->    (true B W -> #)    ->
            (true (A => B) W -> #).
=>F :
      (true A W -> false B W -> #) ->
        (false (A => B) W -> #).
```

```
% box
□T :
      (true A W' -> #) ->
      (true (□ A) W -> #).
□F :
      (w:world false A w -> #) ->
      (false (□ A) W -> #).

% dia
◇T :
      (w:world true A w -> #) ->
      (true (◇ A) W -> #).
◇F :
      (false A W' -> #) ->
      (false (◇ A) W -> #).

% conjunction
&T :
      (true A W -> true B W -> #) ->
        (true (A & B) W -> #).
&F :
      (false A W -> #)   ->   (false B W -> #) ->
             (false (A & B) W -> #).

% falsehood

⊥T : true ⊥ W -> #.

% admissibility of excluded middle (cut)

xm : A:prop W:world (true A W -> #) -> (false A W -> #) -> # -> type.
%mode xm +A +W +D +E -F.

% initial cuts
xt_init : xm A W ([xt] contra xt D) ([xf] E xf) (E D).
xf_init : xm A W ([xt] D xt) ([xf] contra E xf) (D E).

% unused assumptions
xt_unused : xm A W ([x1] contra DT DF) E (contra DT DF).
xf_unused : xm A W D ([x1] contra ET EF) (contra ET EF).

% falsehood. actually falsehood is trivial because a principal
% cut is impossible, and the "commutative" cases aren't even inductive.
xd_ct⊥ : xm A W ([at] ⊥T DD) _ (⊥T DD).
xe_ct⊥ : xm A W _ ([af] ⊥T EE) (⊥T EE).
```

```
% commutative cases.
% note that we need to consider each rule in both D and E, so
% there are four cases for each connective.

% implication
xd_ct=> : xm A W ([at : true A W] =>T
                   ([cf : false C W'] D1 at cf)
                   ([dt : true D W'] D2 at dt) DD) ([af] E af)
                (=>T ([cf : false C W'] F1 cf) ([dt : true D W'] F2 dt) DD)
            <- (cf : false C W' xm A W ([at] D1 at cf) E (F1 cf))
            <- (dt : true D W'  xm A W ([at] D2 at dt) E (F2 dt)).

xe_ct=> : xm A W ([at] DD at) ([af : false A W] =>T
                                ([cf : false C W'] E1 af cf)
                                ([dt : true D W'] E2 af dt) EE)
                (=>T ([cf] F1 cf) ([dt] F2 dt) EE)
            <- (cf : false C W' xm A W DD ([af] E1 af cf) (F1 cf))
            <- (dt : true D W'  xm A W DD ([af] E2 af dt) (F2 dt)).

xd_cf=> : xm A W ([at : true A W] =>F
                   ([ct : true C W'] [df : false D W'] D' at ct df) DD)
                ([af : false A W] E af)
                (=>F ([ct : true C W'][df : false D W'] F ct df) DD)
            <- (ct : true C W' df : false D W'
                  xm A W ([at] D' at ct df) E (F ct df)).

xe_cf=> : xm A W ([at : true A W] DD at)
                ([af : false A W] =>F
                   ([ct : true C W'] [df : false D W'] E' af ct df) EE)
                (=>F ([ct : true C W'][df : false D W'] F ct df) EE)
            <- (ct : true C W' df : false D W'
                  xm A W DD ([af] E' af ct df) (F ct df)).

% box
xd_ct□ : xm A W ([at : true A W] □T ([bt : true B W'] D1 at bt) DD)
                 ([af] E af)
                 (□T ([bt : true B W'] F1 bt) DD)
           <- (bt : true B W' xm A W ([at] D1 at bt) E (F1 bt)).

xe_ct□ : xm A W ([at] D at)
                 ([af : false A W] □T ([bt : true B W'] E1 af bt) EE)
                 (□T ([bt : true B W'] F1 bt) EE)
           <- (bt : true B W' xm A W D ([af] E1 af bt) (F1 bt)).
```

```
xd_cf□ : xm A W ([at : true A W] □F ([w][bf : false B w] D1 at w bf) DD)
                ([af] E af)
                (□F ([w][bf] F1 w bf) DD)
          <- (w : worldbf : false B w
                 xm A W ([at] D1 at w bf) E (F1 w bf)).

xe_cf□ : xm A W ([at] D at)
                ([af] □F ([w][bf : false B w] E1 af w bf) EE)
                (□F ([w][bf] F1 w bf) EE)
          <- (wbf xm A W D ([af] E1 af w bf) (F1 w bf)).

% dia
xd_ct◇ : xm A W ([at : true A W] ◇T ([w][bt] D1 at w bt) DD)
                ([af] E af)
                (◇T ([w][bt] F1 w bt) DD)
          <- (wbt xm A W ([at] D1 at w bt) ([af] E af) (F1 w bt)).

xe_ct◇ : xm A W ([at] D at)
                ([af] ◇T ([w][bt] E1 af w bt) EE)
                (◇T ([w][bt] F1 w bt) EE)
          <- (wbt xm A W ([at] D at) ([af] E1 af w bt) (F1 w bt)).

xd_cf◇ : xm A W ([at] ◇F ([bf : false B W'] D1 at bf) DD)
                ([af] E af)
                (◇F ([bf] F1 bf) DD)
          <- (bf xm A W ([at] D1 at bf) ([af] E af) (F1 bf)).

xe_cf◇ : xm A W ([at] D at)
                ([af] ◇F ([bf : false B W'] E1 af bf) EE)
                (◇F ([bf] F1 bf) EE)
          <- (bf xm A W ([at] D at) ([af] E1 af bf) (F1 bf)).

% conjunction
xd_ct& : xm A W ([at] &T ([ct][dt] D1 at ct dt) DD)
                ([af] E af)
                (&T ([ct][dt] F1 ct dt) DD)
          <- (ctdt xm A W ([at] D1 at ct dt) ([af] E af) (F1 ct dt)).

xe_ct& : xm A W ([at] D at)
                ([af] &T ([ct][dt] E1 af ct dt) EE)
                (&T ([ct][dt] F1 ct dt) EE)
          <- (ctdt xm A W ([at] D at) ([af] E1 af ct dt) (F1 ct dt)).

xd_cf& : xm A W ([at] &F ([cf] D1 at cf) ([df] D2 at df) DD)
                ([af] E af)
                (&F ([cf] F1 cf) ([df] F2 df) DD)
          <- (cf xm A W ([at] D1 at cf) ([af] E af) (F1 cf))
          <- (df xm A W ([at] D2 at df) ([af] E af) (F2 df)).
```

```
xe_cf& : xm A W ([at] D at)
                ([af] &F ([cf] E1 af cf) ([df] E2 af df) EE)
                (&F ([cf] F1 cf) ([df] F2 df) EE)
          <- (cf xm A W ([at] D at) ([af] E1 af cf) (F1 cf))
          <- (df xm A W ([at] D at) ([af] E2 af df) (F2 df)).

% principal cuts. there is one case for each connective;
% a use of the true  rule on A in D, and
% a use of the false rule on A in E.

% implication
x_=> : xm (A => B) W
          ([it : true  (A => B) W] =>T ([af : false A W] D1 it af)
                                       ([bt : true  B W] D2 it bt) it)
          ([if : false (A => B) W] =>F ([at : true  A W]
                                        [bf : false B W] E1 if at bf) if)
          F
        <- (at : true A W bf : false B W
            xm (A => B) W ([it] =>T ([af : false A W] D1 it af)
                                    ([bt] D2 it bt) it)
                          ([if] E1 if at bf) (E1' at bf))
        <- (af : false A W
            xm (A => B) W ([it] D1 it af)
                          ([if] =>F ([at][bf] E1 if at bf) if) (D1' af))
        <- (bt : true B W
            xm (A => B) W ([it] D2 it bt)
                          ([if] =>F ([at][bf] E1 if at bf) if) (D2' bt))
        <- (bf : false B W
            xm A W ([at : true A W] E1' at bf)
                   ([af : false A W] D1' af) (F' bf))
        <- xm B W ([bt] D2' bt) ([bf] F' bf) F.

% box
x_□ : xm (□ A) W ([nt] □T ([at : true A W'] D1 nt at) nt)
                 ([nf] □F ([w][af : false A w] E1 nf w af) nf)
                 F
       <- (w : worldaf : false A w
           xm (□ A) W ([nt] □T ([at] D1 nt at) nt)
                      ([nf] E1 nf w af) (E1' w af))
       <- (at : true A W'
           xm (□ A) W ([nt] D1 nt at)
                      ([nf] □F ([w][af] E1 nf w af) nf) (D1' at))
       <- xm A W' ([at : true A W'] D1' at)
                  ([af : false A W'] E1' W' af) F.

% dia
```

```
x_◇ : xm (◇ A) W ([nt] ◇T ([w][at] D1 nt w at) nt)
                 ([nf] ◇F ([af] E1 nf af) nf)
                 F
       <- (w : worldat : true A w
           xm (◇ A) W ([nt] D1 nt w at)
                      ([nf] ◇F ([af] E1 nf af) nf) (D1' w at))
       <- (af : false A W'
           xm (◇ A) W ([nt] ◇T ([w][at] D1 nt w at) nt)
                      ([nf] E1 nf af) (E1' af))
       <- xm A W' ([at : true A W'] D1' W' at) ([af] E1' af) F.

% conjunction

x_& : xm (A & B) W ([&t] &T ([at][bt] D1 &t at bt) &t)
                   ([&f] &F ([af] E1 &f af) ([bf] E2 &f bf) &f)
                   F
       <- (atbt xm (A & B) W ([&t] D1 &t at bt)
                             ([&f] &F ([af] E1 &f af)
                                      ([bf] E2 &f bf) &f)
                             (D1' at bt))
       <- (af xm (A & B) W ([&t] &T ([at][bt] D1 &t at bt) &t)
                           ([&f] E1 &f af)
                           (E1' af))
       <- (bf xm (A & B) W ([&t] &T ([at][bt] D1 &t at bt) &t)
                           ([&f] E2 &f bf)
                           (E2' bf))
       <- (bt xm A W ([at] D1' at bt) ([af] E1' af) (D1'' bt))
       <- (xm B W ([bt] D1'' bt) ([bf] E2' bf) F).

%block blockw : block w : world.
%block blockt : some A : prop W : world block tt : true  A W.
%block blockf : some A : prop W : world block ff : false A W.

%worlds (blockw | blockt | blockf) (xm A W D E F).

%total A [D E] (xm A W D E F).



%%% Translation from natural deduction to sequent calculus.

% G;D |- A @ W    then   G # D,A
ndseq : A @ W -> (false A W -> #) -> type.
%mode ndseq +D -F.

% conjunction
```

```
ns-&I : ndseq (&I D1 D2) (&F F1 F2)
         <- ndseq D1 F1
         <- ndseq D2 F2.

ns-&E1 : ndseq (&E1 (D1 : A & B @ W))
          ([af] F af)
          <- ndseq D1 ([&f] F2 &f)
          <- (af xm (A & B) W ([&t] &T ([at][bt] contra at af) &t)
                              ([&f] F2 &f) (F af)).

ns-&E2 : ndseq (&E2 (D1 : A & B @ W))
          ([bf] F bf)
          <- ndseq D1 ([&f] F2 &f)
          <- (bf xm (A & B) W ([&t] &T ([at][bt] contra bt bf) &t)
                              ([&f] F2 &f) (F bf)).

% falsehood

ns-⊥E : ndseq (⊥E (D1 : ⊥ @ W))
                ([af] F af)
                <- ndseq D1 ([⊥f] F2 ⊥f)
                <- (af xm (⊥) W ([⊥t] ⊥T ⊥t)
                                  ([⊥f] F2 ⊥f) (F af)).

% implication

ns-=>I : ndseq (=>I ([a : A @ W] D a))
               ([=>f : false (A => B) W]
                =>F ([at : true A W][bf : false B W] F at bf) =>f)
          <- (a : A @ Wat : true A W
              % here's our block
              % think of this as the base case for the
              % assumption we're making.
              ndseq a ([af] contra at af) ->
              ndseq (D a) ([bf] F at bf)).

ns-=>E : ndseq (=>E D1 D2)
               ([bf] F bf)
          <- ndseq D1 ([if] F1 if)
          <- ndseq D2 ([af] F2 af)
          <- (bf xm (A => B) W
                ([it] =>T ([af] F2 af) ([bt] contra bt bf) it)
                ([if] F1 if) (F bf)).

% box
```

```
ns-□I : ndseq (□I [w] D1 w) (□F [w][af : false A w] F w af)
         <- (w ndseq (D1 w) (F w)).

ns-□E : ndseq (□E D1) ([af] F af)
         <- ndseq D1 ([□f] F1 □f)
         <- (af xm (□ A) W ([□t] □T ([at] contra at af) □t)
                             ([□f] F1 □f) (F af)).

ns-□G : ndseq (□G D1) ([□f] F □f)
         <- ndseq D1 ([□f'] F1 □f')
         <- (□f xm (□ A) W'
              ([□t' : true (□ A) W']
                 □F ([w'' : world][af'' : false A w'']
                     (□T ([at'' : true A w''] contra at'' af'') □t')) □f)
              ([□f' : false (□ A) W'] F1 □f')
              (F □f)).

% dia

ns-◇I : ndseq (◇I D) ([◇f] ◇F ([af] F af) ◇f)
         <- ndseq D ([af] F af).

ns-◇E : ndseq (◇E D1 ([w : world][a : A @ w] D2 w a))
              ([cf : false C W] F cf)
         <- ndseq D1 ([◇f] F1 ◇f)
         <- (w' : worlda : A @ w'at : true A w'
              % another use of our block
              ndseq a ([af] contra at af) ->
              ndseq (D2 w' a) ([cf] F2 w' at cf))
         <- (cf : false C W
              xm (◇ A) W ([◇t] ◇T ([w][a] F2 w a cf) ◇t)
                         ([◇f] F1 ◇f) (F cf)).

ns-◇G : ndseq (◇G D) ([◇f : false (◇ A) W] F ◇f)
         <- ndseq D ([◇f' : false (◇ A) W'] F1 ◇f')
         <- (◇f : false (◇ A) W
              xm (◇ A) W'
              ([◇t'] ◇T ([w''][at''] ◇F ([af''] contra at'' af'') ◇f) ◇t')
              ([◇f'] F1 ◇f')
              (F ◇f)).

% we need a way of connecting nd continuation assumptions
% with sequent false assumptions
contfalse : A * W -> false A W -> type.
%mode contfalse +D -F.
```

```
% note contraction: F in output uses af twice
ns-letcc : ndseq (letcc ([ac] D ac)) ([af] F af af)
            <- (ac : A * W af : false A W
                  contfalse ac af ->
                  ndseq (D ac) (F af)).

% note weakening: [cf'] is unused.
ns-throw : ndseq (throw D K) ([cf'] F AF)
            <- ndseq D ([af] F af)
            <- contfalse K AF.

%block blockh : some A:prop W:world
               block a : A @ W at : true A W
                     _ : ndseq a ([af] contra at af).

%block block* : some A:prop W:world
               block ac : A * W af : false A W _ : contfalse ac af.

%worlds (blockw | blockh | block*) (contfalse D F).
%worlds (blockw | blockh | block*) (ndseq D F).

%total D (contfalse D F).
%total D (ndseq D F).

%% Continuation Substitution (excluded middle) for natural deduction
%% This is the price we pay for using A * W instead of hoas for conts

% if   G,x:A@W; D |- M : *
% and  G; D,u:A@W |- N : B
% then G; D       |- [[ x.M/u ]] N : B

xs : (cw A @ W -> c @ w) ->
     (A * W -> B @ W') ->
         (B @ W') -> type.              %name xs U.
%mode xs +D +E -F.

% special: for any term closed wrt the continuation assumption,
%   substitution is the identity. This keeps us from having to
%   treat the case of @ variables, since they are always closed
%   wrt * variables. Without this trick, world subsumption forces
%   cases of xs to infect any later theorem that uses it!

xs-closed : xs D ([a*] E) E.

% falsehood
```

```
xs-⊥E : xs D ([u] ⊥E (EE u)) (⊥E EE')
           <- xs D EE EE'.

% conjunction

xs-&I : xs D ([u] &I (EA u) (EB u)) (&I F1 F2)
         <- xs D EA F1
         <- xs D EB F2.

xs-&E1 : xs D ([u] &E1 (E u)) (&E1 F)
          <- xs D E F.

xs-&E2 : xs D ([u] &E2 (E u)) (&E2 F)
          <- xs D E F.

% implication

xs-=>I : xs D ([u] =>I ([aw : A @ W] E aw u)) (=>I ([aw] F aw))
          <- (aw : A @ W xs D (E aw) (F aw)).

xs-=>E : xs D ([u] =>E (DF u) (DA u)) (=>E FF FA)
          <- xs D DF FF
          <- xs D DA FA.

% box

xs-□I : xs D ([u : A * W] □I ([w] E u w)) (□I ([w] F w))
         <- (w : world xs D ([u] E u w) (F w)).

xs-□G : xs D ([u] □G (E u)) (□G F)
         <- xs D E F.

xs-□E : xs D ([u] □E (E u)) (□E F)
         <- xs D E F.

% possibility

xs-◇E : xs D ([u] (◇E (E1 u) ([w][a] E2 w a u))) (◇E F1 ([w][a] F2 w a))
         <- xs D E1 F1
         <- (w:worldaw: A @ w
               xs D (E2 w aw) (F2 w aw)).

xs-◇I : xs D ([u] ◇I (E u)) (◇I F)
         <- xs D E F.

xs-◇G : xs D ([u] ◇G (E u)) (◇G F)
         <- xs D E F.
```

```
xs-letcc : xs D ([u] letcc ([v] E v u)) (letcc ([v] F v))
         <- (v : B * W' xs D (E v) (F v)).

% throw to different cont
xs-throwmiss : xs D ([u] throw (E u) V) (throw F V)
         <- xs D E F.

% when reaching the throw, pass the term we're throwing
% to D instead.
xs-throwhit  : xs D ([u] throw (E u) u) (D B W' F)
                <- xs D E F.

% world and totality decls come after seqnd, which uses xs.
% they infect each other somewhat, but the worlds decls for
% xs are not substantially different than if it is checked
% alone (there are just some extra unrelated additions)



%%% Translation from Sequent Calculus to Natural Deduction

%       G # D     then   G ; D |- M : *
seqnd :   #         -> (aw a @ w) -> type.  %name seqnd S.
truend :  true A W  -> A @ W -> type.          %name truend T t.
falsend : false A W -> A * W -> type.          %name falsend F f.
%mode seqnd +D -F.
%mode truend +D -F.
%mode falsend +D -F.

% judgmental

sn-contra : seqnd (contra AT AF) ([c : prop][w : world] throw A AC)
             <- truend AT A
             <- falsend AF AC.

% For each connective, the T side is easy -- it just corresponds to
% the elimination rule. The F side requires the "excluded substitution"
% theorem above, and is often quite tricky.

% falsehood

sn-⊥T : seqnd (⊥T FT) ([c][w] ⊥E FT') <- truend FT FT'.

% conjunction
```

```
sn-&T : seqnd (&T ([at][bt] D at bt) T&) ([c][w] F (&E1 N&) (&E2 N&) c w)
         <- (ata
                truend at a ->
             btb
                truend bt b ->
             seqnd (D at bt) ([c][w] F a b c w))
         <- truend T& N&.

sn-&F : seqnd (&F ([af] D1 af) ([bf] D2 bf) (F& : false (A & B) W))
              FF
         <- falsend F& N&
         <- (afa* falsend af a* ->
             seqnd (D1 af) ([c][w] F1 a* c w))
         <- (bfb* falsend bf b* ->
             seqnd (D2 bf) ([c][w] F2 b* c w))
         <- (ccwwb : B @ W
             xs ([c][w] [a] throw (&I a b) N&)
                ([a*] F1 a* cc ww)
                (F1' b cc ww))
         <- (cc : propww
             xs ([c][w] [b] F1' b c w)
                ([b*] F2 b* cc ww)
                (FF cc ww)).

% implication

% actually, implication is not just the elim rule, because
% classical implication is phrased differently.

sn-=>T : seqnd (=>T ([af : false A W] D1 af) ([bt] D2 bt)
                     (T=> : true (A => B) W))
               FF
          <- truend T=> N=>
          <- (afa* falsend af a* ->
              seqnd (D1 af) ([c][w] F1 a* c w))
          <- (btb
                truend bt b ->
              seqnd (D2 bt) ([c][w] F2 b c w))
          <- (ccww
                xs ([c][w] [a] (F2 (=>E N=> a) c w))
                   ([a*] F1 a* cc ww)
                   (FF cc ww)).
```

```
% letcc-free version
sn-=>F : seqnd (=>F ([at : true A W][bf : false B W] D at bf) F=>)
               ([c][w] throw (=>I [a] FZ a B W) N=> )
          <- falsend F=> N=>
          <- (ata
                truend  at a  ->
              bfb* falsend bf b* ->
              seqnd (D at bf) ([c][w] F1 a b* c w))
          <- (a : A @ W
              ccww
                xs ([c][w] [b] throw (=>I [a-unused : A @ W] b) N=>)
                   ([b*] F1 a b* cc ww)
                   (FZ a cc ww)).

% also include simpler letcc version
% u : A=>B |- throw (\x:A . letcc v : b* in (IH) end) to u
sn-=>F-letcc :
        seqnd (=>F ([at : true A W][bf : false B W] D at bf) F=>)
              ([c][w] throw (=>I [a] letcc [b*] F1 a b* B W) N=>)
          <- falsend F=> N=>
          <- (ata
                truend  at a  ->
              bfb* falsend bf b* ->
              seqnd (D at bf) ([c][w] F1 a b* c w)).

% box

sn-□T : seqnd (□T ([at' : true A W'] D at') T□)
                  ([c][w] F (□E (□G N□)) c w)
         <- (at'a'
                truend at' a' ->
             seqnd (D at') ([c][w] F a' c w))
         <- truend T□ N□.

% this is the only necessary letcc in the proof
% u : []A * w |- throw (box w'. letcc v:A*w' in (IH) end) to u
sn-□F : seqnd (□F ([w' : world][af : false A w'] D w' af) F□)
              ([c][w] (throw (□I [w'] letcc ([a*'] F w' a*' A w')) N□))
         <- falsend F□ N□
         <- (w' : worldaf' : false A w'a*' : A * w'
                falsend af' a*' ->
             seqnd (D w' af') ([c][w] F w' a*' c w)).

% dia
% x : ◇A@w |- let dia <y,w'> = get<w>x in IH end
sn-◇T : seqnd (◇T ([w'][at] D w' at) T◇)
         ([c][w] ◇E (◇G N◇) ([w'][a'] F w' a' c w))
         <- truend T◇ N◇
         <- (w'at : true A w'a' : A @ w'
               truend at a' ->
             seqnd (D w' at) ([c][w] F w' a' c w)).

sn-◇F : seqnd (◇F ([af'] D af') F◇)
         FF
         <- falsend F◇ N◇
         <- (af' : false A W'a*' : A * W'
               falsend af' a*' ->
             seqnd (D af') ([c][w] F1 a*' c w))
         <- (ccww
               xs ([c][w] [a'] (throw (◇G (◇I a')) N◇))
                  ([a*'] F1 a*' cc ww)
                  (FF cc ww)).

% finally, world and totality verification for our theorems.

% for xs
%block blocku : some A : prop W : world
                block u : A * W.

%block blocka : some A : prop W : world
                block u : A @ W.

% for seqnd
%block blocknt : some A:prop W:world
                 block at:true A W a:A @ W
                  t:truend at a.
%block blocknf : some A:prop W:world
                 block af:false A W a:A * W t:falsend af a.

%block blockp : block a:prop.

%worlds (blockw | blocka | blocknt | blocknf | blocku | blockp)
     (xs D E F) (seqnd D F) (truend D F) (falsend D F).

%total E (xs D E F).

%total (D T F) (seqnd D FF) (truend T N) (falsend F M).
```